ARMY RESEARCH LABORATORY

Project Focus: A Study of Virtual Proving Ground Software Architecture Requirements

Geoffrey C. Sauerborn
Kenneth G. Smith
Alan W. Scramlin
Robert R. Shankle
Robert W. Gauss
Weiqun Zhou
Toney R. Perkins
Patrick E. Corcoran
John A. Weller
Richard W. Marvel
Joseph P. Schimminger

ARL-TR-1429

SEPTEMBER 1997

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-1429 September 1997

Project Focus: A Study of Virtual Proving Ground Software Architecture Requirements

Geoffrey C. Sauerborn
Weapons & Materials Research Directorate, ARL

Kenneth G. Smith
Information Science & Technology Directorate, ARL

Alan W. Scramlin
Robert R. Shankle
Robert W. Gauss
Aberdeen Test Center

Weiqun Zhou
Computer Science Corporation

Toney R. Perkins
Patrick E. Corcoran
Weapons & Materials Research Directorate, ARL

John A. Weller
US Army Tank-Automotive and Armaments Command

Richard W. Marvel
SFA, Inc. (Frederick Manufacturing Division)

Joseph P. Schimminger
Aberdeen Test Center

Abstract

The virtual proving ground (VPG) is a concept being developed within the U.S. Army Test and Evaluation Command to harness the power of state-of-the-art sophisticated modeling and simulation technologies to augment and enhance test and evaluation in support of product acquisition. VPG is a cohesive and comprehensive capability for testing concepts, virtual prototypes, hardware prototypes, subsystems, and full systems. A broad, far-reaching, and diverse set of capabilities is envisioned within the VPG. Critical to the successful implementation of the VPG is an architecture able to support or enable those capabilities. A major function of the VPG architecture will be to integrate dissimilar heterogeneous engineering level models and simulations of prototype and production hardware and the synthetic environments in which they operate.

In 1996, the U.S. Army Aberdeen Test Center and the U.S. Army Research Laboratory jointly conducted "Project Focus" to help determine the architectural requirements that support the VPG concept. This report contains a description of Project Focus and the architectural requirements that resulted from it.

ACKNOWLEDGMENTS

The authors wish to gratefully acknowledge the contributions made by the following persons: to Mr. Paul Oxenberg and Major Michael Smith for their leadership, encouragement, and extended freedom which allowed meaningful exploration; to Scott Redding, Tim Hart, and John Reilly for their additional software development "behind the scenes"; to Alan Etzel, Doug Gipprich, Drew Moore for lending their consultative expertise and "brain storming."

INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK

LIST OF FIGURES

INTENTIONALLY LEFT BLANK

LIST OF TABLES

INTENTIONALLY LEFT BLANK

PROJECT FOCUS: A STUDY OF VIRTUAL PROVING GROUND SOFTWARE ARCHITECTURE REQUIREMENTS

1. INTRODUCTION

The virtual proving ground (VPG) is a concept being developed within the U.S. Army Test and Evaluation Command (TECOM) to harness the power of state-of-the-art sophisticated models and simulations (M&S) technologies to augment and enhance test and evaluation in support of product acquisition. VPG is a cohesive and comprehensive capability for testing concepts, virtual prototypes, hardware prototypes, subsystems, and full systems. A broad, far-reaching, and diverse set of capabilities is envisioned within the VPG. Critical to the successful implementation of the VPG is an architecture able to support or enable those capabilities. A major function of the VPG architecture will be to integrate dissimilar heterogeneous engineering level models and simulations of prototype and production hardware and the synthetic environments in which they operate.

In 1996, the U.S. Army Aberdeen Test Center (ATC) and the U.S. Army Research Laboratory (ARL) jointly conducted a project to determine specific architectural requirements and software features that support the VPG concept. This report describes that project and the concluding architectural requirements identified to date. This effort was called "Project Focus."

2. PURPOSE

The purpose of Project Focus was to distill the existing list of software architectural requirements, identify additional requirements, and focus on those requirements that are both necessary and sufficient for conducting tests and evaluations in the VPG. Specifically, it was decided to construct a prototype VPG system that would enable a tester to duplicate the process of conducting a simulated test on the VPG. The scope covered all phases of the process: (a) virtual test planning, (b) test design and execution, (c) virtual instrumentation, (d) data collection, and (e) post processing data analysis (including incorporation of ground truth data). Design and development for each phase of the prototype VPG were conducted with the intent of identifying and classifying issues and requirements germane to the VPG, particularly its (software) architecture.

3. PROJECT FOCUS

The goals of Project Focus (namely, determining architectural requirements for fuzing [possibly disparate] simulations in a virtual test environment) were addressed by building upon architectural research conducted before Project Focus. This research was the result of a technology program annex (TPA) between TECOM and ARL. Results of this earlier work provided the design and implementation of a low level set of "core" software services (the "VPG core"). The VPG core was greatly expanded and enhanced during Project Focus.

In the conceptual implementation of the VPG, system software (and underlying software architecture) would support a test director's design and conduct of a test. To the greatest reasonable extent, the same processes and procedures used to conduct a physical test would be implemented in the virtual test. A general scenario followed by the test director would be as shown in Table 1.

Table 1. General Physical Tester's Test Process

1. Review the test item performance requirements.
2. Determine which types of tests could prove or disprove these requirements.
3. Design and execute the tests.
4. Analyze the results.

Steps 1 and 2 are assisted by documents called test operations procedures (TOPs). TOPs outline in a general way the types of tests and procedures that generally could be used to determine whether the equipment meets requirement objectives.

In the VPG, the tester would follow the steps in Table 2.

Table 2. Project Focus Virtual Tester's Test Process

1. Review on-line TOPs applicable to the virtual equipment.
2. Determine which types of simulations would be appropriate to test the applicable requirements.
3. Run those simulations.
4. Analyze the results.

For Project Focus, Steps 1 and 2 were integrated so that just by selecting a particular TOP, the tester was ensured that simulations applicable to that TOP would be available (and simulations that may not be appropriate were hidden). After initial selection of a TOP, the user could select parameters to vary within those applications, run the applications, and analyze the results.

These steps sound simple enough, but there is much that goes on behind the scenes to support this functionality. Where are data sets (if any) that could be used as input to the application? How are these data transported to the place where the simulation can use them? Where do disparate simulations place their "output" and how should this output be analyzed? What if the tester wished to search for different kinds of data or simulations to use? These and many other mechanisms must exist somewhere; therefore, it makes sense have them available as overall low level VPG system functions since they are likely to be used repeatedly among many VPG applications. Therefore, these functions were organized beneath the tester's interface in an area call the "VPG core." Procedures that were determined to be "core" to the type of operations required by all or many VPG applications were placed in the VPG core.

In the following sections, we explain the overall architecture structure and organization of the VPG core.

3.1 <u>Overall Architectural Structure</u>

The prototype architecture that was used for Project Focus was designed with several premises in mind. The first premise is that the role of the architecture is to provide the means for compatibility and interoperability of models and simulation in a synthetic testing environment. The second premise is that the diversity of models and simulations expected to be used in the VPG (particularly considering those already existing or being developed) is such that a single all-encompassing architecture is unrealistic. The third premise is that even if such an architecture existed or could be designed, the success of VPG would then hinge on the entire user community "getting on board" and using or adapting to this architecture. Hence, an approach was taken to try to design a "low level" architecture or infrastructure that functions below the operational level commonly viewed by most M&Ss today. In short, the approach taken is to try to describe M&Ss operationally, that is, describe them in terms of their input and output (I/O) characteristics and requirements and do so in such a way that testers/evaluators, using sophisticated VPG software tools, can configure and construct test scenarios (virtual tests) from existing and available M&Ss in as seamless and automated a fashion as possible.

In Project Focus, the architecture employed can be viewed in terms of an infrastructure (the low level architecture) and a superstructure (see Figure 1). The infrastructure provides the basic functionality for the integration, interactability, and interoperability of M&Ss. The superstructure extends that capability by applying it to a particular implementation of a system--in this case, the VPG. In object-oriented parlance, the superstructure would be a class derived from the infra-structure base class and from which an instance, VPG, is created.

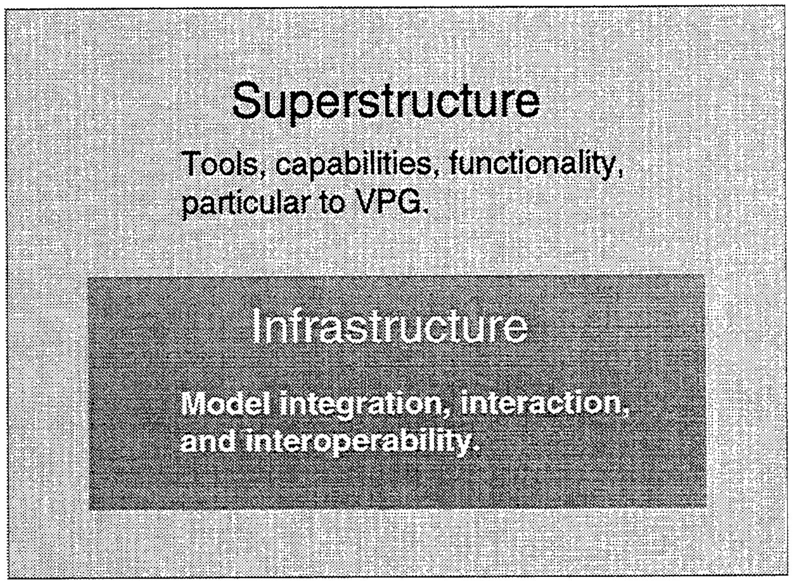

Superstructure
Tools, capabilities, functionality, particular to VPG.
Infrastructure
Model integration, interaction, and interoperability.

Figure 1. Project Focus VPG Architecture.

The Project Focus prototype architecture is centered around the M&Ss' I/O description concept. A language, called a model description language (MDL), is being developed to sufficiently describe M&S I/O characteristics and requirements. The notion is that all elements available to the VPG (models, simulations, simulators, test procedures, test data and even VPG "tools" [defined later in this section]) are described by an MDL file. These MDL files provide the basis for determining what can be done in VPG and how to do it.

3.1.1 *The Architectural Core*

The core architecture, as the infrastructure is commonly referred to, is a prototype architecture being developed around this MDL approach. It is designed to operate in a "plug-and-play" fashion. Where and when they fit, test procedures, guidelines, ground truth data, and M&Ss to be used/evaluated in the VPG can all be individually plugged into the VPG and be immediately available for interaction with other such elements. This is made possible by the core architecture being able to read, understand, and manipulate (often with control from the user) the MDLs of the elements.

First, "element" needs to be defined. To VPG, an element is any object that can be operated on, by, or within the VPG. Software models or simulations that emulate or represent actual (or conceptual) systems or subsystems are considered to be elements. Hardware or soldier-in-the-loop simulators are viewed as elements. Data generated by these M&Ss and simulators, as well as data collected from actual field tests, are all viewed as elements. Finally, the software programs (known as VPG tools) that add user functionality to VPG (such as programs for constructing, configuring, and executing test scenarios [comprised of elements] in the VPG) are themselves viewed as elements.

From the core architecture's perspective, the VPG is simply a collection of elements, some of which are VPG tools that work with the other elements. Tools enable the user to do virtual testing of the elements that represent concepts, virtual prototypes, hardware prototypes, subsystems, and full systems. To this end, the core architecture really provides two services: (a) a collection point for accessing all the elements, and (b) utilities for accessing, interpreting, and manipulating the elements.

In practice, it is often difficult and unnecessary to clearly delineate between layers such as infrastructure and superstructure. For implementation purposes, a slightly different view of the architecture is discussed. This view takes a more layered look at the architecture developed for Project Focus. Figure 2 depicts this view, which shows two main areas: (a) the architectural core and (b) the supporting components. Each of these is discussed in the following sections. If these two views are merged, the infrastructure (from Figure 1) would be shown to encompass most of the VPG application programmer's interface (API), VPG database (DB) API, and DB plus one or two supporting components tools. The remaining tools and uncovered portions of the VPG API, DB API, and VPG DB would represent those attributes or behaviors comprising the superstructure.

The core architecture is comprised of three layers (VPG API, DB API, and VPG DB) which cooperatively (a) serve as a collection point for accessing all the VPG elements, and (b) provide utilities for accessing, interpreting, and manipulating those elements. The collection point for accessing the VPG elements is referred to as the VPG DB. The routines for accessing, interpreting, and manipulating those elements reside in the two API layers. These three layers are discussed in the next two subsections.

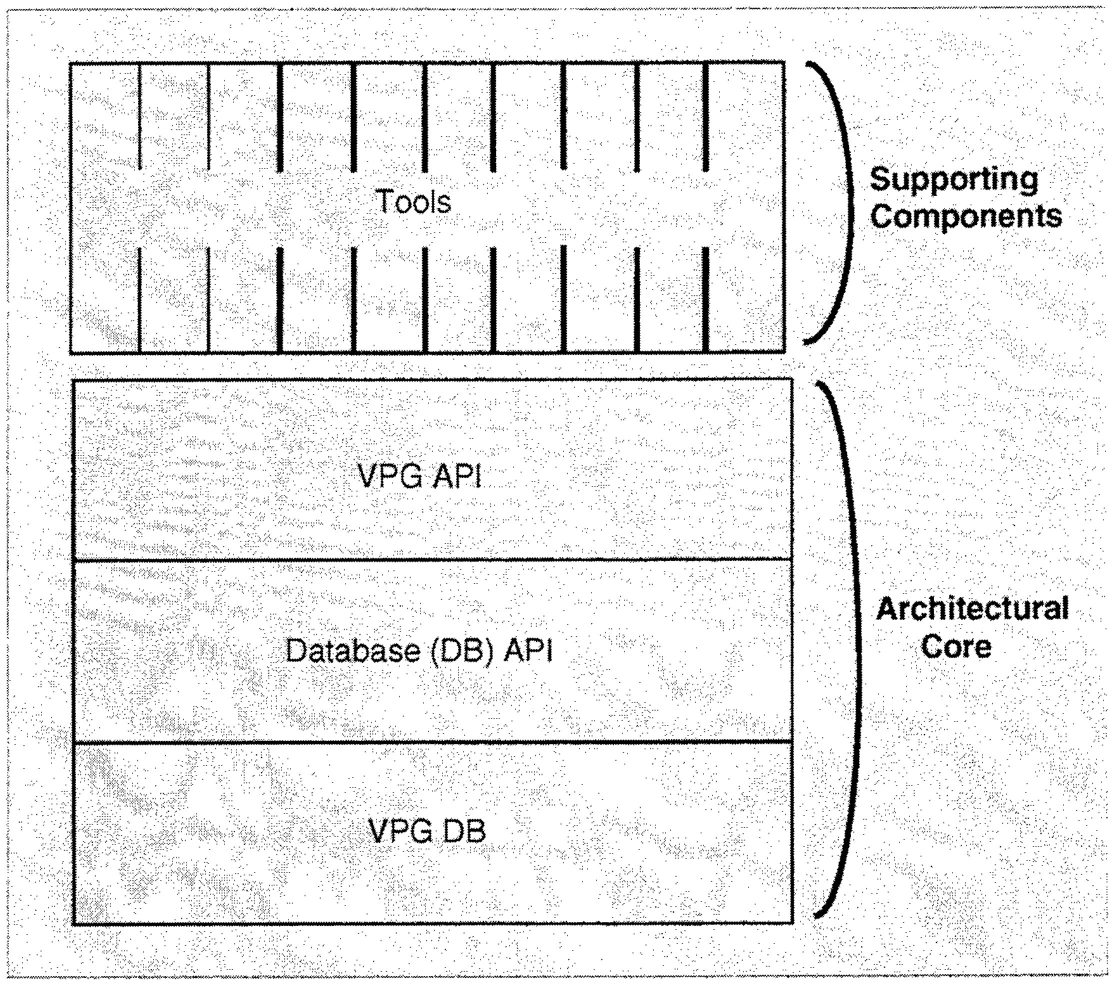


Figure 2. Implementation View of Project Focus VPG Architecture.

3.1.1.1 VPG Core: VPG DB and DB API

The lowest level layer (and fundamental to all layers) is the VPG database layer (VPG DB). The database layer's purpose, not surprisingly, is to serve as a general database. However, in an organizational sense, the database is only used to store information about elements (applications and their associated data sets [inputs and outputs]). These data are not the actual applications or data themselves but are "Meta-Data" (references to the actual data). This information serves as a basis for determining where to find an application, what type of data is needed to run it, what type of data it produces, etc. These facts are pieced together by VPG tools. By parsing these data, an appropriate tool could determine, for example, which applications are prerequisites to other applications (because output produce by one is required as input by another).

The VPG DB API consists of API calls that are used to initialize and connect to the database, define data records, store and retrieve data, and many other functions common to general database usage. Use of the API allows the actual database program (which is doing all the grunt work) and its details to be hidden from the applications and other layers that use it. Another advantage to using an API is that the database underneath could be replaced without affecting the applications that depend on it. Appendix A outlines the service calls available within the API DB layer.

3.1.1.2 VPG Core: VPG API

VPG API functions serve as the interface between application programs (supporting components seen in Figure 2) and the architectural core. The philosophical design for VPG is to build within this layer all procedures that operate on VPG data objects and are needed by the supporting components. This implies that it is unnecessary for a tool to directly access the DB API. This frees tools from the concern of low level database structures and allows a more abstract data object concept to be used.

Creating a more abstract view of data objects and concepts was based on VPG's intended purpose (namely, test and evaluation in a virtual environment).

Creating a more abstract view of data objects and concepts was based on VPG's intened purpose (namely, test and evaluation in a virtual enviroment). In order to translate proceedures taken by a tester into a virtual environment, we must translate the tester's actions into base software functionality. Some of the base software functionality that (in one form or another) must lie underneth these actions would be knowledge of how to make a simulation do what the tester has in mind. Also, associations need to be maintained between the data needed to run a simulation subject and the environment description the tester wishes to impose. That is, a knowledge base must be maintained to keep track of "which switches to turn on" in a simulation and the data required to "feed" the simulation in order to get it to respond to the test conditions the operator had in mind.

Not required but certainly worth having is a configuration service to the tester, the recording of the particular simulations used, their initial conditions, and (possibly abstracted) outputs (i.e., analysis). These data should be maintained as a cohesive unit, and the underlying architectural services should support that.

To implement these base VPG API functions, certain data objects were defined in Table 3.

The philosophical approach is to subdivide the VPG API layer into logical sections, each of which is especially designed to service a particular abstract data object or service. These objects were supported by software library functions for creating, destroying, editing, and manipulating the objects in a manner that would support the concept of a virtual test scenario (see Table 2).

Table 3. Objects Supporting the Conceptual Test Process

Object	Description
tool (or MDL)	A description of data, models, or simulations. (Tools are very similar to the model description language [MDLs] objects first discussed in Section 3.1).
scenModel	An instance of a tool, for use in a scenario (also called a "scenario model").
scen	Scenario. A collection of simulations (scenario models) run during a particular test case.

The VPG API is designed to be expandable. Whenever a new set of functions (a library) was determined to be necessary, it could be easily added to the VPG API layer. (Syntactically, it was agreed to prologue VPG API system calls with "vpg_" followed by an acronym alluding to the object or function serviced by that library. (For example, all software functions manipulating the "tool" object of Table 3, would be named "vpg_tool_something" [e.g., vpg_tool_create(), vpg_tool_destroy(), etc.].) Some of the VPG API libraries completed during Project Focus are described in Table 4. (See Appendix B for a short description of each software procedure from the VPG API libraries.)

3.1.2 *Supporting Components*

Supporting components differ from VPG "external" applications. (Some examples of external applications might be tape storage/retrieval utilities, interactive conferencing tools, spread-sheets, documenting applications, etc.) External applications have general purpose utility, and while that utility could be enhanced by making these applications "VPG aware," it is not required.

VPG supporting components, on the other hand, have to be aware of the VPG environment to perform their function. (An example of a supporting component would be an application to edit MDL objects. An MDL editor would need to be able to retrieve, change, and store these VPG objects and hence would have to [at least indirectly] use the VPG API.) Supporting components tend to be fundamental to many other VPG test procedures and processes.

Table 4. A Subset of VPG API Libraries

Library	Description
vpg_tool	In the VPG, a tool describes an object referencing an executable program that can be started (or launched) by VPG but which is not compiled directly into the VPG core. (That is, the tool itself is not part of the VPG architectural core.) Tools could be simulations, word processors, and other general purpose applications. As mentioned, tool objects are not the actual applications themselves but are "Meta Data" (which refer to them). Procedures within the vpg_tool library created, destroyed, edited, and examined tool objects.
vpg_scenModel	Basically, a "scenModel" is a copy (or instance) of the tool object. The copy is created for use in a test scenario. By using an instance of the tool and not the original, the information can be customized for a particular test scenario without corrupting the original "vpg_tool." Functions within the vpg_scenModel library create, destroy, edit, and examine scenario model objects.
vpg_scen	A scenario is a collection of specific scenModel objects (and their data sets) executed during a virtual test exercise. The vpg_scen library is used to create, destroy, or retrieve scenarios, add or remove scenModel objects from the scenario, etc.
vpg_launchTool	The launch tool section is used to start stand-alone programs that run outside VPG. Launching tools is really the job of VPG daemons. (A daemon is a program that lies dormant until triggered by a certain event. After execution, daemons usually revert to their "dormant" state.) This section of the API determines which daemon is responsible for launching the tool and then it sends a "launch" message to that daemon with all the necessary information required by the tool. By using VPG daemons within the VPG architecture to start programs (tools), the Tester is able maintain control of tools even when run remotely on distant computers.
vpg_err	This is a generalized error-reporting and exception-handling library. Though it can be used by supporting components, mostly it is used by other VPG libraries.
vpg_printf	The print library portion of the VPG API allows the control of the amount and redirection of various messages generated internally by the VPG core. Many messages can be generated during the course of building, executing, and analyzing virtual tests and are useful for debugging or tracking. These routines allow these messages to be channeled to an inconspicuous place until needed.

Another way of viewing supporting components is as “plug-in” tools. That is, even though they are aware of the VPG environment, they can be treated as external stand-alone tools in the sense that they can be described by an MDL (tool) object. The advantage to doing this is that they may now be launched as one would start (or stop) other applications (from a “main” VPG control panel or graphic user interface [GUI]). Also, because information about them is stored in the VPG DB, newly created components have the potential to be immediately available to all VPG users.

As mentioned, supporting components tend to be fundamental to many other applications and VPG processes. Two noteworthy examples are (a) architecture user interface mechanisms, and (b) simulated environment data management. These components (while not a part of the “core” architecture) are so basic to most applications that it was thought to experiment with their functionality early in the prototype VPG to determine how best to integrate them within the VPG. These two subject areas are covered next. In subsection 3.1.2.1, we discuss approaches to data-providing services, and in 3.1.2.2, we examine some user interface approaches.

3.1.2.1 Database Integration (supplying simulations with commonly available data)

All simulations have an initial state. Most often, this state is not “hard coded” into the simulation algorithms but read as data. In the next two subsections, we discuss data-providing services (servers). The first (subsection 3.1.2.1.1) examines a case for providing terrain information (an input data component very common in ground vehicle simulations). In subsection 3.1.2.1.2, we explore more general data server concepts.

3.1.2.1.1 Terrain Database

Because much of ATC’s testing involves ground vehicles (although not exclusively ground vehicles), an essential requirement to many VPG simulations is accurate terrain-related information. The approach used to meet this requirement is to provide a specific terrain data service--the terrain database (TDB). The TDB is tailored to provide real-world terrain-related information for simulations. The following subsection gives a brief description of the TDB server (TDS). Figure 3 outlines the TDS’s design structure.

Layer (1) of Figure 3 represents the client for the TDS. The client can be any program that uses the TDB API to communicate with the TDS. The TDB API is portrayed in Layer (2) of the same figure. It provides the communication and protocol procedures required to “talk” with the server application (3). The client program (1) will embed these API procedures into its design. At compile time, the client is linked with the TDB API object library. Note. The client program does not have to be a stand-alone application. It could actually be part of a large service structure. For example, the VPG core's VPG API layer (see Figure 2) will eventually add the TDB API as one of its services.

```
+--------------+
|              |
|    Client    |  (1)
|              |
+--------------+
|   TDB API    |  (2)
+--------------+
       ^
       |
       |
       v
+--------------+
|              |
|  TDB Server  |  (3)
|              |
+--------------+
       ^
       |
       |
       v
+--------------+
|              |
| TDB Services |  (4)
|              |
+--------------+
```

Figure 3. Terrain Database Server (TDS) Design.

The TDS is shown in Layer (3) of Figure 3. This layer has two functions: it (a) parses and attempts to service client requests and (b) acts as a cache to the geographic information system (GIS). Requests that require terrain database regeneration, culling, translation, or many other direct GIS procedures normally are computationally intensive. Service delays can be avoided by having the TDS keep track of data queries already requested and then provide those (cached) results. If the requested data are not available (in the cache), the server contacts the TDB services in order to generate the requested data.

TDB services are displayed in Layer (4) of Figure 3. The TDB service layer provides a command interface to the GIS engine. Table 5 displays the commands currently available.

Table 5. TDS Service Commands

COMMAND	DESCRIPTION
TDS_STATUS	Return the current status of the TDS, such as "Idle," or the current name of the current command/ARC/INFO Macro Language (AML) being processed. This command also returns a list of real time servers running and the computer-operating system processes (process IDs) they are supporting.
TDS_SUBMIT	usage: TDS_SUBMIT [batch file | message] {output database filename} Submit batch file to the server to supply a database. See format for batch file (see Figure 4).
TDS_DISCONNECT	Break connection to the server.
TDS_HELP	Provide help about commands.
TDS_HUMAN1	Start AML menu to create files and/or perform maintenance
TDS_AVAILABLE	TDS will keep track of available terrain databases and paths. When TDS receives the "TDS_AVAILABLE" command, it will list the available files, with a unique ID and description of each. The description will include dimensions, spheroid, datum, projection, units, and attribute types. For a path, it will also list speed and frequency of sample.
TDS_GET_FILE fileID	When the TDS_AVAILABLE command is given, a list is returned of available terrain databases and paths. Each file will have a unique fileID. Use TDS_GET_FILE to request a specific file.
TDS_INFO	usage: TDS_INFO x,y {,z} Get the attributes for x,y,{z}, in terrain database specified by "TDS_SUBMIT" or "TDS_GET_FILE". If z is missing, it will return z also.

The TDB services module will create new databases based upon the information provided with the "TDS_SUBMIT" command. Figure 4 displays available commands understood when submitted as the batch file (or message) for the "TDS_SUBMIT". Note. If there is more than one option for a parameter, then the first listed is the default.

```
Spheroid ( WGS 84 | Clark 1866 )
Datum (NAD 83 | NAD27)
Projection (UTM)
Units (Meters | FEET | Miles | KM)

Database type: (3D Terrain | Vehicle path | Point Data)

If Database type is 3D terrain: {
   Specify Southwest and Northeast corners of database
         (East Min, North Min, East Max, Northing Max)
}

If Database type is a Vehicle path: {
   Specify the speed of vehicle (15 Km/s)
   Specify the frequency of sample (1Hz)
   Specify path to follow (Premade | File of Easting, Northing  points)
}

If Database type is a Point: {
   Specify Easting and Northing of point:
}

Number of attributes: (1)
Attributes: (Elevation)
   (another example:
   Number of attributes: 4
   Attributes:  Elevation, ITD code, color, surface type)

Output file format:  (DWB, S1000, ARL_path, Open_Flight, textfile)
```

Figure 4. "TDS_SUBMIT" Command Syntax.

3.1.2.1.1.1 Populating the Terrain Database with Source Data

Source data used in Project Focus test scenarios were generated from a highly detailed survey of the U.S. Army ATC "H-field" firing range. The data's accuracy was $\pm$ 0.1 meter elevation for each given longitude/latitude point. These data were measured every 1 to 10 m apart, depending upon the terrain profile (i.e., if a section of a surveyed road was straight, fewer data points were needed, but many points were collected to accurately describe a curve). As an example, when measuring ATC's H-field firing range, more than 5,300 survey points were collected; most of these were points on road edges. Open areas between roads were interpolated, based upon the road edges.

Once survey data were collected, they were used to generate an AutoCAD™ database showing the connectivity of the points. This database was exported to a Drawing eXchange Format (DXF™) American standard code for information exchange (ASCII) file. This provided a drawing of the road edges, building locations, and shore lines (Aberdeen Proving Ground is situated on the shores of the Chesapeake Bay). This DXF™ file and the original data points file were then loaded into a commercial GIS database called ARC/INFO™. Once read into ARC/INFO, a triangulated irregular network (TIN) was created using the DXF™ layer to denote the features (roads, buildings, etc.).

AML was used to create programs that exported the desired formats needed by simulations (e.g., ARL's M1A1 fire control model and the Tank Automotive Research Development and Engineering Center's [TARDEC's] hull motion model). Another AML program exported to an ASCII file which could be read by a commercial tool called Designer's Workbench™ (DWB). DWB was used to create the visual TDB. This TDB is featured in ATC's Stealth application (a tool for visually rendering a vehicle traversing the VPG).

3.1.2.1.1.2 Applying the TDB to Project Focus

As mentioned, the GIS system (ARC/INFO) was used to export data for several applications in various formats. Specifically, Table 6 displays a sample portion of the ASCII files required by the M1A1 fire control simulation. These data represented the path traveled by the test vehicle.

Table 6. Example of ARL Fire Control "Path" Input

0.050,	388493.26,	4354743.82,	5.77
0.100,	388493.53,	4354743.90,	5.78
0.150,	388493.80,	4354743.97,	5.78
0.200,	388494.07,	4354744.05,	5.79
0.250,	388494.34,	4354744.12,	5.79
0.300,	388494.61,	4354744.20,	5.79
0.350,	388494.88,	4354744.27,	5.80
0.400,	388495.15,	4354744.35,	5.80
0.450,	388495.42,	4354744.42,	5.80
0.500,	388495.69,	4354744.50,	5.81
.			
.			
.etc...			

Column headings (not included in file) were time (seconds), X (meters), Y (meters), Z (meters), respectively, based in a local Cartesian coordinate system (or NAD83 datum, Grid 18).

Menu-driven programs were written in ARC/INFO, which allowed the user to generate a vehicle path based on the simulation's requirements. These requirements were vehicle speed, VPG terrain sampling rate, and points roughly describing the vehicle path. A backdrop of H-field was provided for placement of points. Once the user roughly described the path, the points were then smoothed with a cubic spline algorithm to generate a continuous path with points taken at the desired frequency for the given speed. A set frequency of 20 Hz was used (that is 20 sample points along the path per second). Two fire control test vehicle paths were generated, one with the vehicle going 25 mph and one at 12.5 mph (over the same path). In addition, a second path was generated to simulate the moving target. The target vehicle started 3000 m away from the test vehicle and traveled 20 mph toward it. (A program was written to ensure that the starting points of the shooting and target vehicles were 3000 m apart.) The target path was described in the same format as the fire control test vehicle (see Table 6).

The TARDEC hull motion model required the description of its simulated vehicle's path in a different format. An example of this format appears in Table 7.

Table 7. TARDEC Hull Motion Model Path Description

```
1,388493.26329,4354743.82248,5.773
2,388493.53239,4354743.89752,5.777
3,388493.80148,4354743.97256,5.782
4,388494.07057,4354744.04758,5.786
5,388494.33975,4354744.12259,5.790
6,388494.60887,4354744.19756,5.794
7,388494.87801,4354744.27250,5.797
8,388495.14724,4354744.34743,5.801
9,388495.41639,4354744.42228,5.805
10,388495.68552,4354744.49707,5.809
.
.
.
```

The column headings (not included in the file) were time index (no units), X (meters), Y (meters), Z (Elevation) (in meters).

Both ARL and TARDEC used the same source path information, but the data formats differed. ARL required the first column to be in seconds; therefore, the time index was divided by the sampling frequency (20 Hz). TARDEC and ARL were provided the same source data for the test (shooting) vehicle and target vehicle. TARDEC required an additional file to compute roll; therefore, a "grid" file was generated to allow roll calculations. The grid file was generated in ARC/INFO by using the TIN of H-field. A rectangular grid was created with equally spaced points based from this TIN. Only elevation was provided at each "grid" point (one per line). Using this data set, TARDEC was able to calculate its simulated vehicle's roll.

This was the extent of terrain data needs for Project Focus. Future improvements include a terrain database server (TDS) that will allow data to be automatically extracted (with very little human intervention) from the GIS database and provided to simulations.

3.1.2.1.2 Ground Truth Database

"Ground truth" is the combination of experience gained by the material tester community, test procedures, and collected test data. In a software architecture sense, capturing test community experience is the most difficult of these three. In Project Focus, this was addressed through the use of TOPs. In addition, in the future it could be advantageous to have lessons learned, on-line test reports, and other forms of tester's knowledge base available. Test procedures were used by integrating software that addressed those procedures. An example of how TOPs were addressed and used is explained further in Section 3.2. The final and most voluminous portion of ground truth is the set of collected and analyzed test data.

There have been various approaches toward ground truth. For instance, TECOM has previously conceptualized ground truth into classes of data sets, as shown in Table 8.

We do not necessarily agree with this taxonomy since there is some overlap (e.g., time space position really is just a sub class of performance data). However, whatever the final classifications become, the eventual goal is for these data to be stored in the ground truth library. The ground truth library is envisioned as a distributed set of databases. Data collected shall be for a diverse set of systems and components (over a wide spectrum of tests). Therefore, the types, frequency, and quality of these data shall vary greatly in both content and quality. Thus, an essential component for the ground truth library will be a data dictionary that clearly describes and categorizes the data. Also, each data set should include a knowledge base archiving the data's source and other information that allows potential users to qualitatively assess the data's fitness for their particular purpose.

Table 8. Proposed TECOM Ground Truth Classes

Class	Meaning
Time Space Position	Position, velocity, orientation, etc., relative to an initial reference point over the course of time (tracking data).
External Environmental Factors	Weather, road conditions, other factors that will influence the test item and results.
Performance Data	Test item parameters such as weapon accuracy, and failure rates. (The exact nature of data collected depends on the test item and requirements.)
Validation and Verification	This class of data is collected for the specific purpose of validating and verifying a model and/or simulation.
Lethality and Vulnerability	Data that can be used to support calculation of lethality or vulnerability assessments. These could be system signatures which can add or subtract from a system's detectability (thus, vulnerability).

The ground truth library is an eventual goal in TECOM's vision for the VPG. Our more immediate objective (during Project Focus) was to examine how to best incorporate ground truth into the virtual testing process and to do this in as automated a fashion as makes sense. This would serve to scrutinize proposed approaches toward handling ground truth data from both the databasing side (data server) and from the simulation application side (data client). Herein lies a philosophical question mainly concerning how to best capture data for the future. Concerning ground truth data, (a) is the interface to VPG supposed to be an interactive session or (b) is the user/client supposed to already know what data are available and explicitly request them? We think the answer lies in both being true. Testers should be able to browse through various classes and examples of data available in the ground truth library. Results of this data search could influence which tests are conducted and how.

3.1.2.1.2.1 Ground Truth Data Used for Simulation Input

In Project Focus, a virtual fire control test was being applied to the VPG. Therefore, it was decided to use fire control ground truth data for both input to the simulations and as output (for post processing comparative analysis). For our simulated test, the goal was to replicate as faithfully as possible initial test conditions from an actual test and compare the real and virtual test results. It was thought that we would extract the paths followed by the actual test vehicles. This path would then be overlaid on top of VPG terrain (see 3.1.2.1.1 Terrain

Database) to extract the virtual test terrain profile. This profile would then be fed into the simulations.

We searched for historical ground truth data that matched the type of simulated test we were to conduct. However, the state of the available data confirmed several of our concerns. Namely, many data items that could have been used were missing from the existing ground truth tabulation. Of those that were available, many were in a state that limited their use. The greatest shortfall was the inadequate correlation between geometric field measurements and a geographic model. Specifically, what was needed was a correlation between actual field geographic measurements and VPG terrain stored in the GIS database (see 3.1.2.1.1 Terrain Database). Such a correlation could not be made because the historical record provided a localized Cartesian coordinate system. This coordinate system remained consistent for all measured test trials; this was good. However, one could not determine the exact points "on the earth" where the origin and major axis of this local coordinate system lay. Through trial and error, we were able to come close but not close enough to justify using historical "ground truth" as input to the vehicle path.

3.1.2.1.2.2 Data Used for Comparative Analysis With Simulation Output

Project Focus's simulated results and the historical test results could be related but only in an aggregate sense. Too many variations in physical test conditions (for the historical measurements) and assumptions made for initial simulation conditions (for the simulated results) make side-by-side comparison impractical. In addition, as stated earlier, a major consideration was that we were unable to correlate the test vehicle path with the simulated environment. Furthermore, physical test outputs (the historical record) were the result of transformations and filtering made on raw measurements. We can only assume that proper transformations were made and any round-off errors are well below the overall noise.

Another influence embedded within historical test measurements are characteristics of the test instrumentation used to conduct the measurements. If these test instruments significantly influenced the historical measurements, then those same influences should be accounted for (see 3.2.1.3 ATC Through-Sight Video Simulator).

3.1.2.2 User Interface

We have just discussed a major supporting component in the VPG environment (data services). Another supporting component also of extreme importance is the user interface. The architectural core provides a general application programming interface into the fundamental VPG procedures. Using this API, an experienced programmer could design and build his or her own interface. The interface could be as simple as a single program that executes a specific task. Of far greater use would be a more general purpose interface allowing the user to conduct a variety of commands and actions. Two pervasively popular interfaces are the command line interface (CLI) and GUI. The CLI is text based requiring the user to type commands interactively or submit prepared commands (batch files). A GUI interactively presents the user with visual menus, icons, buttons, and other selectable widgets. (Even with a GUI, however, eventually some type of keyed input is almost always necessary.) During Project Focus, prototypes for both types of user interfaces were demonstrated (CLI and GUI). These are discussed in the next two subsections.

3.1.2.2.1 VPG Graphic User Interface (GUI)

The VPG GUI demonstrates the simplicity with which user interfaces can be created when an underlying service call structure (the VPG API) is available. The VPG main GUI (seen in Figure 5) is a simple X-Windows "Xt" application and consists of four buttons and a few pull-down menus. The "tools" pull-down menu can be used to start the VPG database (if it is not already running). The four buttons seen are linked to VPG database records. (Specifically, the database records are MDLs that describe the application to be launched.) These are "soft links" and can be changed easily (e.g., by modifying an initialization text file or by adding a command line argument).

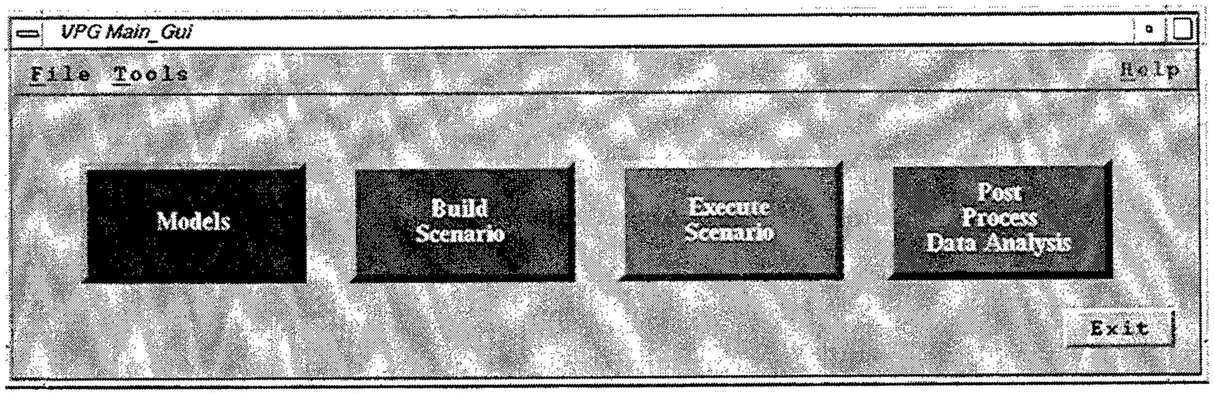


Figure 5. VPG Main GUI.

The "models" button ensures that all scenario models (see Table 3) created by the user are loaded in the database. The "build scenario" button launches an application where the tester may combine models and data into a logical test scenario. The "execute scenario" button launches a very simple application that queries the VPG database for the latest scenario created by the user and then executes that scenario. "Post process data analysis" executes an analysis tool to examine test scenario results. For Project Focus, this consisted of plotting various measures of effectiveness from simulation output. Also, the post processing done was specific to the test scenario executed during Project Focus (see subsection 3.2.2.3 Post Processing and Analysis: [A Manually Intensive Task]). However, the point being made concerning the VPG main GUI is that it outlines some of the basic building steps taken in the virtual testing process. Any number of tools with greater sophistication can be easily attached to the GUI.

Generally speaking, the VPG core is designed to run on POSIX operating compliant operating systems. The VPG main GUI is UNIX X-Windows based and also normally would run on that same type of computer platform. In the next subsection, we describe another approach (using world wide web [WWW] techniques) to the user interface, which allows the GUI to be even more platform independent.

3.1.2.2.2 HTTP Server

The hypertext transport protocol (HTTP) server is a command line user interface which can also serve as an interface to a hypertext markup language (HTML) WWW browser interface to the VPG, and this is precisely what it was used for during Project Focus.

By default, this user interface (called "vpg_command") expects input from a WWW client, but it can also be run interactively as a CLI. As a CLI, the server accepts typed user commands from the keyboard (or a file). This is a generalized command processor interface to the VPG API. However, as with the VPG GUI, the VPG database server must be running first. This is because the database is the only means by which test scenarios and scenario components can be accessed. Table 9 lists vpg_command's implemented capabilities at the time of Project Focus.

Table 9. "vpg_command" Project Focus Implemented Capabilities

Define (build) a Scenario Model.
Define (build) a Scenario.
Delete a Tool.
Delete a Scenario Model.
Delete a Scenario.
List Tools.
List Scenario Models.
List Scenarios.
Execute a Scenario.

The syntax for each of these commands is given in Appendix C. The following is an example of how the VPG user could execute one of these commands (specifically, the build-a-test scenario). First, in interactive mode, the vpg_command is started by typing

```
vpg_command.exe -i
```

The "-i" option causes the command interface to operate interactively. (Recall that by default, vpg_command looks for input from an HTTP server and not from standard input [the keyboard].) Figure 6 displays the syntax for building a scenario. (Note. The control-D [^D] ends the standard [keyboard] input stream. Indentation and line breaks are insignificant.)

```
        VPG_COMMAND = SCENARIOMODEL_DEFINE
         BEGIN = SCENARIOMODEL_DEFINE

          NAME = "this is my new test scenario's name"
               #  (text after any "#" symbol is ignored)
               #
               #  the key that follows must be a TOOL's key (an MDL's key)
               #   in the VPG database.
               #
          KEY  = KEYverSep96_385_0x803f279a_511_842878033_20423_2

         END = SCENARIOMODEL_DEFINE

^D
```

Figure 6. Sample vpg_command Language Syntax.

When run under an HTTP server (e.g., not interactively), vpg_command processes commands as an HTTP/1.0 "Post" query method. What this means is that commands are received in the syntax of "variable name=value" pairs. This explains the proliferation of "=" symbols woven into the vpg_command language syntax.

Simple common gateway interface (CGI) scripts can then be constructed, which call vpg_command to process the Post query. For example, Figure 7 displays a CGI script (written in UNIX Bourne shell syntax).

```
#!/bin/sh
VPG_DB_HOST=www.vpg.db.com

echo 'Content-type: text/html'
echo
echo '<HTML>'
echo '<body>'
echo '<H1><P ALIGN=CENTER>VPG COMMAND(S) SUBMITTED</H1>'
echo '<UL>'

./vpg_command.exe -d ${VPG_DB_HOST}

echo '</UL>'
echo '</body>'
```

Figure 7. Example CGI Script Accessing vpg_command via WWW Interfaces.

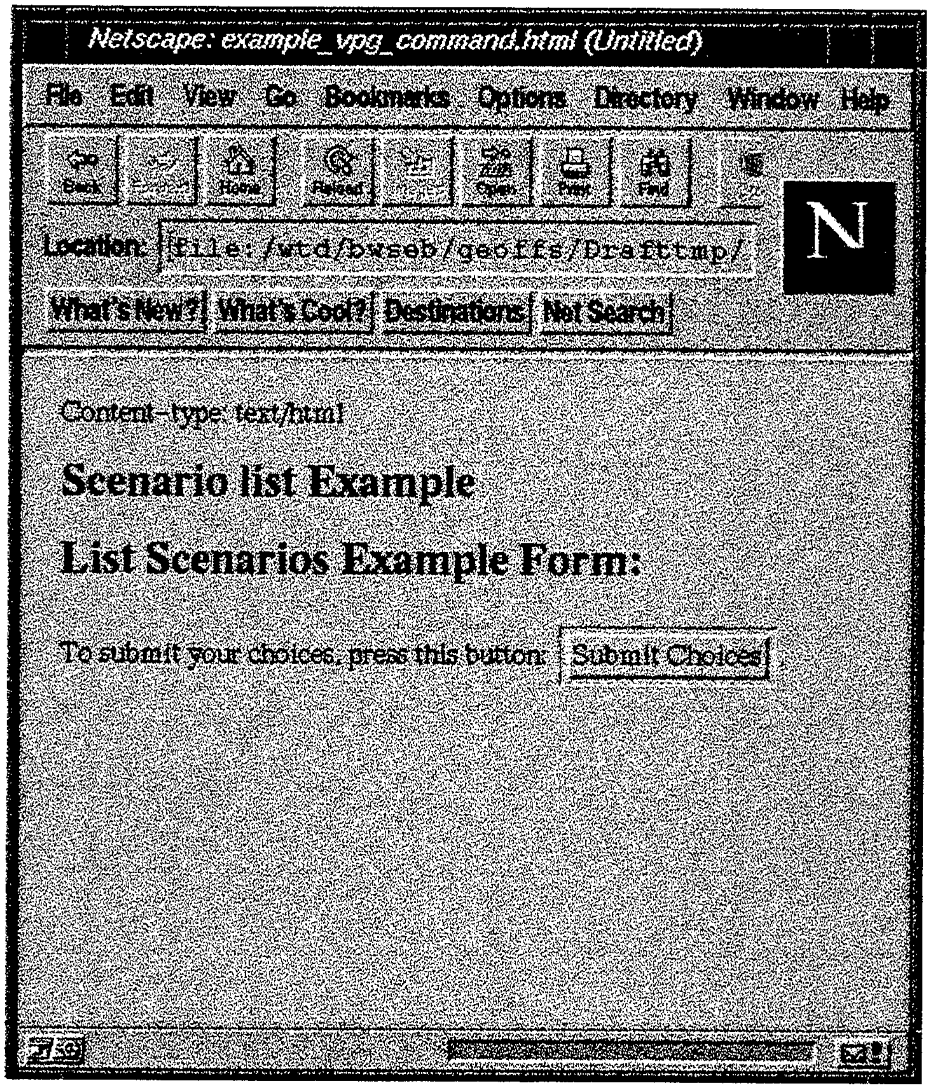
Suppose we name the script of Figure 7 "vpg_command.cgi." Using "vpg_command.cgi," we will now be able to process any vpg_command (that is sent to the HTTP server via the Post query format). By default, vpg_command expects the VPG database server to be running on the same host on which it is being run. The "-d" option can be used to specify a different host. Note that in this example, the fictitious host "www.vpg.db.com" represents the internet protocol (IP) address of the host machine that is running the VPG database. To send commands with the Post query format, do the following:

Now that we have established a CGI script that is ready to accept commands from the WWW, the next step is to construct a "web page" to do just that. The following is a sample Post query HTML page that could be used to submit a command to the WWW address running the HTTP server. Let us assume that the WWW address running the HTTP server is "www.vpg.com" and the CGI script is saved as "/cgi-bin/vpg_command.cgi" on that server. Figure 8 is an example Post query HTML page that could be used to send a command ("list saved scenarios") to vpg_command.

Note. This HTML page could be run from a WWW browser residing on any computer. When read by a browser, it would appear similar to Figure 9. The user would select the "submit" button, and the built-in command ("VPG_COMMAND = SCENARIO_LIST") would be sent to the vpg_command interface.

```
Content-type:  text/html

<HTML><body><H1>Scenario list Example</H1>
<FORM METHOD="POST"
ACTION="http://www.vpg.com/cgi-bin/vpg_command.cgi">
<H2>List Scenarios Example Form:</H2>

<INPUT TYPE="hidden" NAME=VPG_COMMAND
VALUE=SCENARIO_LIST >

To submit your choices, press this button: <INPUT TYPE="submit"
VALUE="Submit Choices">. <P>

</FORM></UL></body></HTML>
```

Figure 8. Example List Scenarios Command "Web Page."


Netscape: example_vpg_command.html (Untitled)
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Location: File:/wtd/bwseb/geoffs/Drafttmp/
What's New? What's Cool? Destinations Net Search
N
Submit Choices

Content-type: text/html

Scenario list Example

List Scenarios Example Form:

To submit your choices, press this button:

Figure 9. Sample VPG Command Through a "Web" Browser.

If vpg_command were run locally (and interactively), the same results could be achieved by typing "vpg_command.exe -i -d www.vpg.db.com" to start vpg_command and then interactively typing the command

```
VPG_COMMAND = SCENARIO_LIST
```

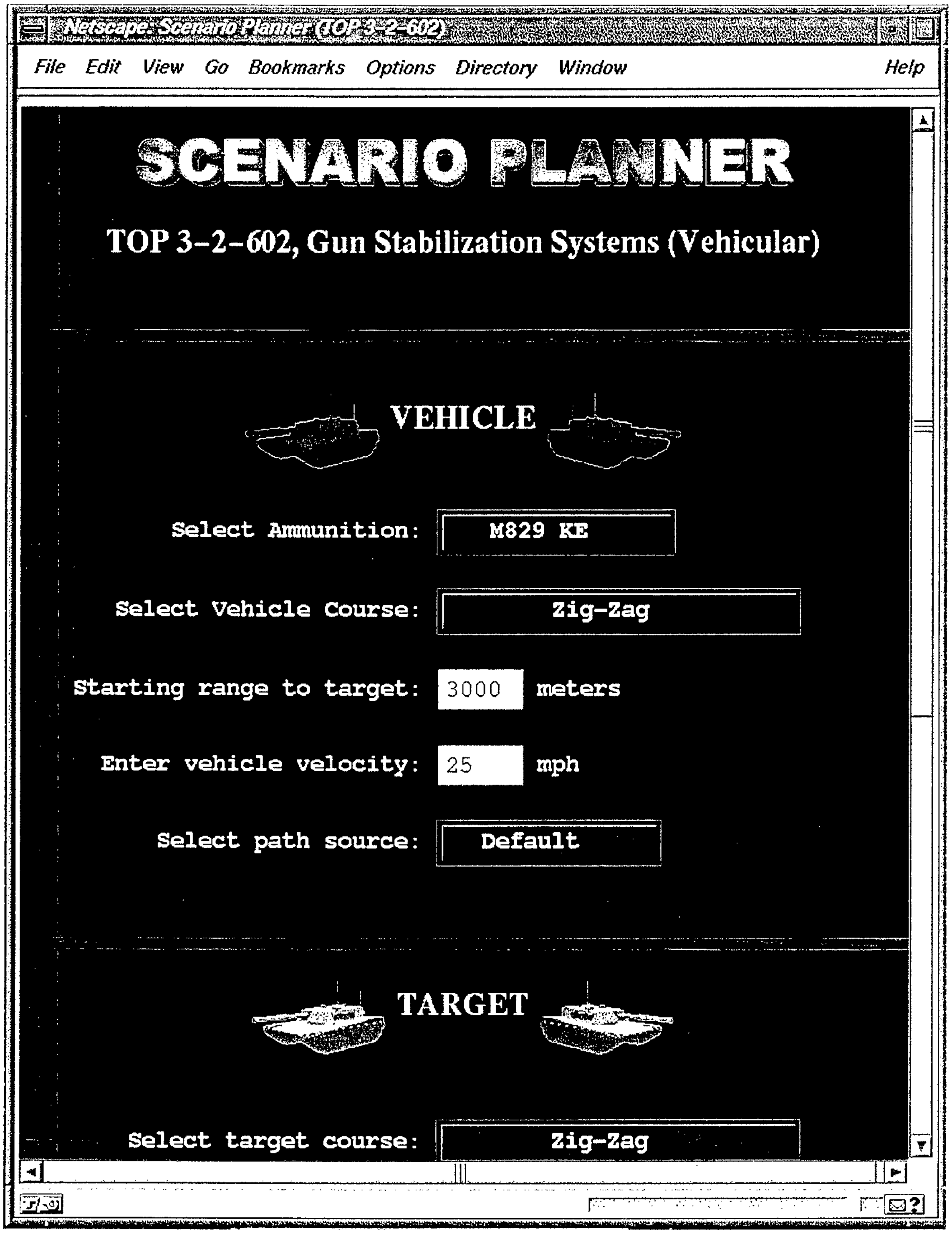
In the same manner, other web pages could be constructed to build an entire GUI interface on top of the CLI program. The next section (3.2) demonstrates numerous examples of these "WWW GUI" interfaces (notably subsection 3.2.2.2). The execution phase consists strictly of a series of HTML pages used to make Post queries of CGI scripts (which in turn call the vpg_command program). The great advantage is that all interactive processing could be done from just about any computer with a web browser.

3.2. Project Focus: A Pseudo VPG Application Test Project (tying disparate simulations together)

To prove how well the architecture design works in practice, a trial case was designed. This section of the report describes the Project Focus trial scenario and ideally, how the user would interface with the VPG GUIs. Some of the more interesting (and hidden to the user) capabilities of the VPG architecture were not implementable within the constraints of this trial; these are pointed out later in subsection 3.2.2.2.

The trial case was designed to mimic the general process taken by a test when conducting a ground vehicle fire control weapon system test. Ideally, for a given system or component, general test operations procedures are outlined in a document call a TOP. The TOP describes the types of measurements that need to be made to test the performance of the components or systems in question.

Through the VPG GUI and WWW interfaces, the test director was able to select which TOP he was interested in following. After TOP selection, the user was presented with parameters that could address the performance characteristics outlined in the TOP. This was accomplished by varying parameters relating to the tests to be conducted. In the case of the fire control test, parameters describing the test vehicle, target, and other test conditions were available as shown in Figure 10. Associated with these parameters were simulations that could model the test(s) required by the TOP. However, the specific simulations themselves remained hidden for the moment, allowing the test engineer to concentrate on the environmental conditions.


Netscape: Scenario Planner (TOP 3-2-602)
File Edit View Go Bookmarks Options Directory Window Help

SCENARIO PLANNER

TOP 3-2-602, Gun Stabilization Systems (Vehicular)

VEHICLE

Select Ammunition: M829 KE

Select Vehicle Course: Zig-Zag

Starting range to target: 3000 meters

Enter vehicle velocity: 25 mph

Select path source: Default

TARGET

Select target course: Zig-Zag

Figure 10. Parameters Associated With a Particular TOP.

Following this, the test director could determine which computer models were available to simulate these parameters in the fire control test scenario. In our trial case, the test director could select from a number of models. For each permutation of models selected, there was only one topology that would make sense as far as how the models related to one another for the purpose of analysis. The final selected topology was visually displayed for the test director. (Figure 11 shows the visual display when the ARL tank fire control simulation, TARDEC hull motion simulation, and ATC through-sight video simulator were all selected for the virtual test.) Simulation results for these simulations fed into a plotting program (GNU Plot) which displayed test results. Determining which results to display is a result of having selected a particular TOP (in this case, the gun stabilization systems [vehicular] TOP).

3.2.1 *Participating Project Focus Simulations*

As mentioned, the trial case was designed to prove how well the architectural design works with "real simulations," particularly engineering level simulations. The models used in these experiments were a tank fire control simulation, a tank hull motion simulation, a through-sight video simulator (developed at ATC), and a hull motion simulation developed at TARDEC. Details of these simulations follow in the next three subsections.

3.2.1.1 ARL Tank Fire Control Simulation

The origin of the M1A1 engineering simulation began in the early 1970's when a family of computer codes called HIT-PRO (HIT PRObability) was developed to evaluate the performance of combat vehicles and their weapon systems. Several versions of HIT-PRO were developed over the years to model different combat vehicles, and in 1985, under contract to ARL (then the Army's Ballistic Research Laboratory), the Ordnance Systems Division of the General Electric Company, the developer of HIT-PRO, adapted it to model an M1A1 tank.

The M1A1 simulation contains detailed engineering models of the suspension, hull, and turret. The fire control system includes the ballistic computer, the turret azimuth and gun elevation control systems, and gunner azimuth and elevation tracking models. The azimuth and elevation gunner models differ but they are universal models in that these same models are used regardless of target or M1A1 motion. In reality, the gunner is adaptive whereby he would tailor his response to the target and M1A1 motion.

Inputs to the model are the M1A1 design characteristics, ammunition characteristics, target motion, M1A1 motion, and the terrain over which the M1A1 moves. Although many outputs are available from the simulation, the primary outputs are the azimuth and elevation tracking errors and lead angles. Tracking error is defined as the angular displacement of the reticle measured with respect to the target. Lead angle is the angular displacement of the gun measured with respect to the target. These primary outputs from the simulation are the same as those measured during actual fire control tests conducted at ATC.

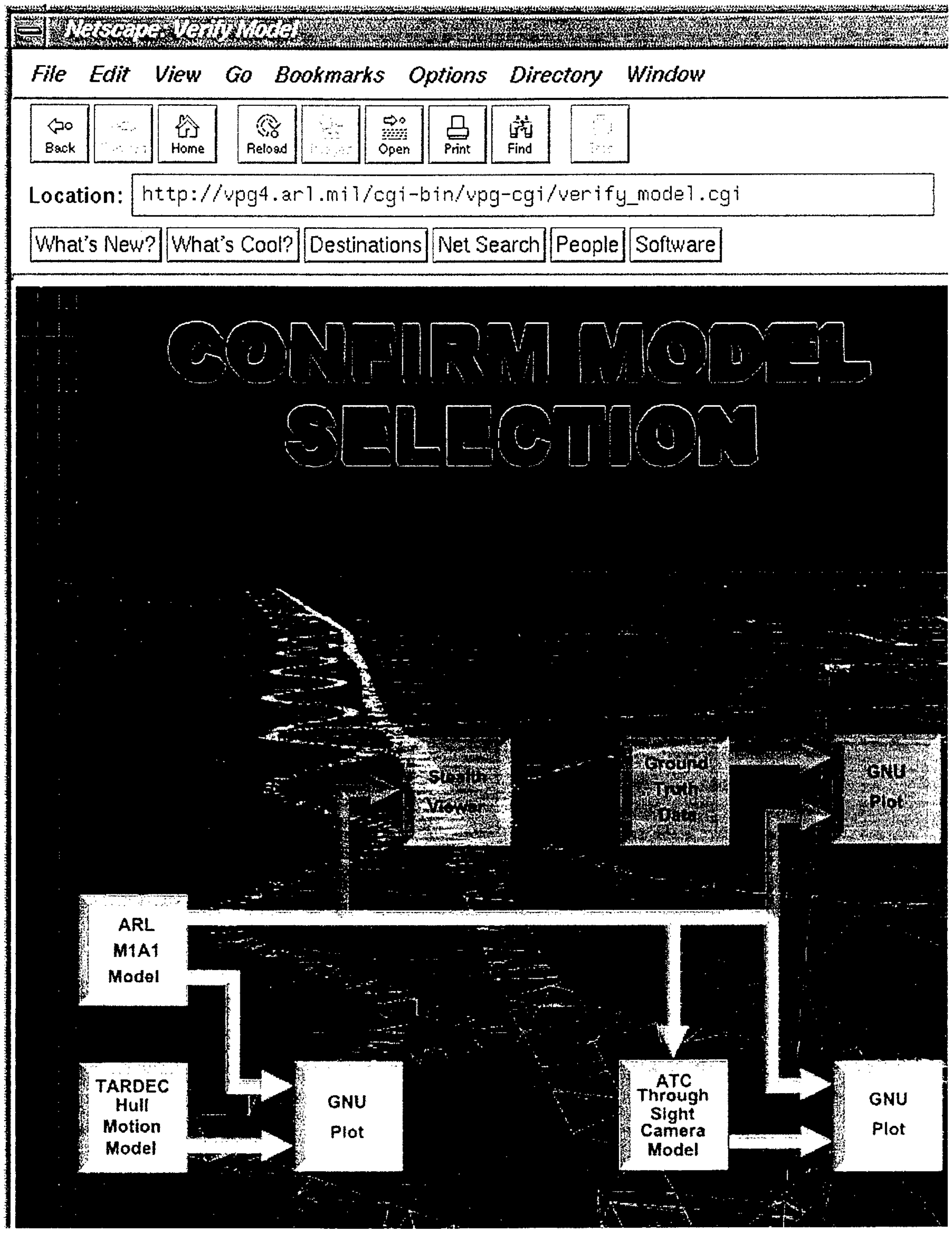


Figure 11. A Selected Test Scenario Topology.

Before the M1A1 engineering simulation was used in the VPG, the simulation was reviewed in detail to ensure its correctness. This review showed that there were numerous approximations, and many variables were improperly initialized. These approximations were replaced with the actual equations and all the variables were properly initialized. Several errors were also found and corrected.

To ensure the fidelity of the reviewed and updated M1A1 engineering simulation, its primary outputs were compared to actual test results. Rather than conduct a test to obtain the data necessary to validate the simulation, data from a test conducted in the 1992-1993 time frame were used. The purpose of this previous testing was to determine the feasibility of incorporating an auto-tracker into the M1A1. Two basic scenarios were considered: a stationary M1A1 engaging a maneuvering target, and a maneuvering M1A1 engaging a stationary target. Manual tracking as well as auto-tracking engagements were included in this test. Only the manual tracking trials were used for this comparison analysis.

The first testing scenario was designed to evaluate the performance of a stationary M1A1 against both air and ground targets maneuvering at ranges of 1, 2, 3, and 4 kilometers. When engaging ground targets, the gunners simulated firing kinetic energy (KE) and high explosive antitank (HEAT) rounds, and when engaging air targets, the gunners simulated firing KE and training rounds. The training round served as a surrogate for the multipurpose antitank (MPAT) round since the flight times of these two rounds are similar. This testing was conducted in ATC's moving target simulator (MTS).

The maneuvering M1A1 versus stationary target testing scenario was conducted at ATC's H-field zig-zag course. As the name implies, the course is basically sinusoidal. The M1A1-to-target initial range was approximately 3 kilometers, and the M1A1 traveled over this course at speeds of 12.5 and 25.0 mph. Only KE rounds were considered for this scenario.

Along with the M1A1 and target motion, the primary signals measured during all this testing were tracking errors and lead angles. Similar outputs from the simulation were compared to these measurements for the same M1A1-target conditions when gunners manually tracked the target.

Results of this comparison showed that the simulation does a fair job of duplicating the tracking errors but most importantly, does a very good job of duplicating the lead angles. Since the lead angles showed very good agreement, the simulation can be used with a high degree of confidence to estimate a given projectile's probability of hit for the types of engagements considered in this comparison. To improve the gunner tracking models, future plans call for developing adaptive models using system identification techniques to determine the gunner's response.

Two reports[1] have been written documenting in detail the M1A1 engineering simulation, and the results of the comparison between its outputs and the outputs of the actual system.

3.2.1.2 TARDEC Hull Motion Simulation

One of the envisioned capabilities for the VPG is to be able to draw upon various models during a simulated test scenario. In this manner, one could use models within the context of what they are best suited to simulate. (For our test scenario, the ARL tank fire control simulation was used to simulate the fire control performance, while the hull motion model simulated forces on the hull.)

Use of the hull motion simulation, developed by the U.S. Army TARDEC, demonstrated another VPG envisioned capability, that of remote distributed computing. Because of its configuration, the TARDEC hull motion simulation had to be run at a separate site. In such an event, the VPG would request the remote site to launch the specified simulation (the TARDEC hull motion simulation), specify or provide require inputs, and retrieve the simulation results. (Of course, proper user permissions and protocol would first be established before this could occur.) Because of time constraints, this remote execution step was not automated for the hull motion model during Project Focus.

The TARDEC hull motion simulation is a physics-based, multi-body, three-dimensional dynamics model of a combat vehicle's turret, hull, track, and suspension subsystems. This model is an engineering simulation that predicts the motion histories of these subsystems and the histories of reaction loads acting on these subsystems and the ground. The TARDEC hull motion simulation was built using the commercially available software, Dynamic Analysis and Design System (DADS). DADS is a general purpose modeling and simulation method for determining the spatial, transient-dynamic response of controlled, articulated multi-body mechanical systems to excitation by irregular external and internal forces. The methodology consists of a library of subroutines defining primitive rigid body, kinematic joint, control element,

[1] Corcoran, P.E., and Perkins, T.R., "A Comparison of ARL's M1A1 Engineering Simulation Results With Actual Test Results," ARL-MR-347, March 1997.
Perkins, T.R., and Corcoran, P.E., "M1A1 Engineering Simulation for the Virtual Proving Ground: Description User's Guide," ARL-MR-360, May 1997.

and force-building blocks that can be combined in numerous ways to assemble complex system models to the desired level of detail and accuracy.

The hull motion simulation contains detailed models of the suspension, track, and ground-track interface. It has been used to simulate a variety of operating scenarios of a combat vehicle including traversal of proving ground courses, stationary firing, firing while moving, target tracking (static and moving), obstacle crossing, transportation scenarios (bridge crossing, trailer and rail transport), and others. Motion histories from this model are routinely used as input to drive the controllers of the motion base simulators at TARDEC.

The TARDEC hull motion simulation model inputs include

- Mass, center of gravity, and mass distribution (moments and products of inertia) for each body of the vehicle (i.e., hull, turret, gun).
- Suspension characteristics such as torsion bar stiffness, shock characteristics, road arm and road wheel masses and inertias, track mass and stiffness, and suspension assembly geometry.
- Desired speed (constant or varying), path to be steered, and terrain to be traversed.
- Terrain/obstacle profile and soil properties.
- Turret traversal and gun elevation angles (constant or varying).
- Gun firing time(s) and loads (time varying).

For Project Focus, the hull motion simulation outputs were the hull northing, easting, altitude, roll, pitch, and yaw. Velocities and accelerations of these quantities were also available.

3.2.1.3 ATC Through-Sight Video Simulator

To achieve correspondence with physical testing, it has been determined that modeling of the test instrumentation is essential to the VPG effort. The objective of virtual instrumentation is modeling the effect of instrumentation on the physical parameters that are measured during test. It encompasses characterization of instrumentation in terms of bandwidth, accuracy, and test item loading. Virtual instrumentation is also concerned with providing insight in determining the requirements of the potential integration of instrumentation into future systems to be tested, in such a manner that minimizes system loading.

was analyzed to develop a computer model that characterizes the instrumentation's transfer function. The bandwidth of a through-sight imaging system used to measure tracking error was derived. The results of the analysis were then simulated through computer implementation of the measurement process. This computer simulation was applied to the tracking error signals generated by the ARL M1A1 fire control simulation.

3.2.2 *Results From Applying VPG Concepts to These Simulations*

In Section 3, we outlined the hypothetical steps made when conducting a test in the VPG. In the following subsections (3.2.2.1, 3.2.2.2, 3.2.2.3), we walk through those steps and see how they were implemented during Project Focus.

3.2.2.1 **Preparing Heterogeneous Simulations for the VPG** (a manually intensive task)

We mentioned (in Section 3) that the on-line TOPs already contained knowledge of which models applied to the TOP in question. This might be advantageous to the user by providing assurance that only simulations applicable to that TOP are available (and simulations that may not be appropriate were hidden). However, when designing the test, it makes sense to allow the tester to be able to add or subtract from the list of available models. This functionality was not included in the demonstration because of the manually intensive process needed to fold new simulations into the VPG database. At this time, at least manual procedures are required as outlined in Table 10.

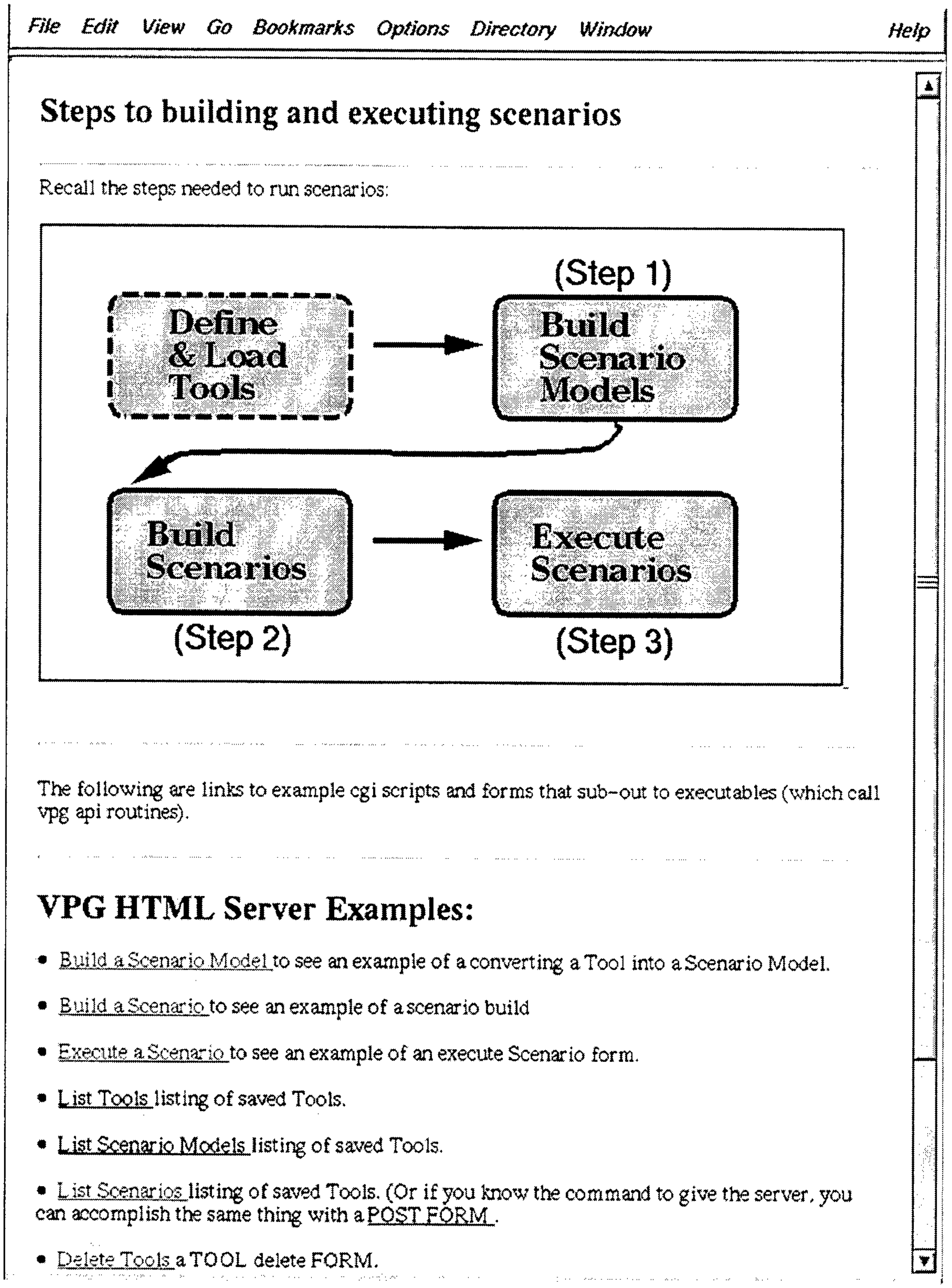
The VPG core provides all the root (core) services needed to find simulations and data and combine them any number of ways. However, preparing the VPG core to do this (populating the VPG database with data) is a manual process. This situation can be greatly improved with user-friendly tools to assist and debug this process. Such tools are needed to make the VPG test scenario process a reality that is usable "to the masses."

3.2.2.2 The Execution Phase

The execution phase consists of connecting the models and data into logical test scenarios and executing that scenario. Once simulations and their required data have been accurately described and inserted into the VPG database, connecting those pieces together and executing the resulting scenarios is simplistic. This simplicity is a result of tools that were completed during Project Focus. Figure 12 illustrates the build and execution phase. (Figure 12 is a snapshot taken of an actual web-based tool that uses the VPG HTTP command server described in subsection 3.1.2.2.2.)

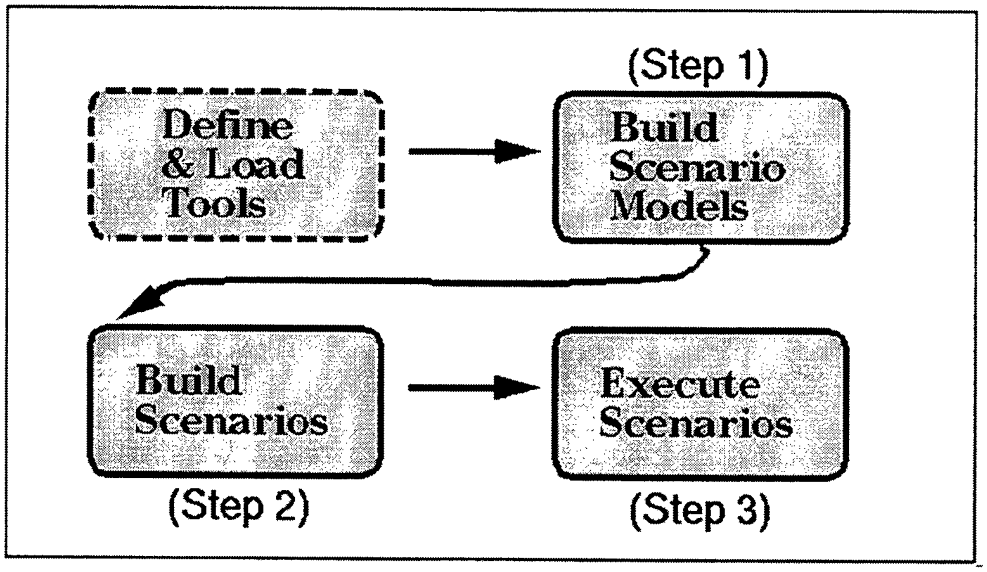
Table 10. Steps Required to Insert External Applications (simulations) into the VPG Core Demonstration Architecture

Procedure	Description
1. Insert references to the simulation into VPG database	All references to the simulation are described by MDL records in the VPG database. Building MDL records is a manually intensive process (although portions of MDL build process could be automated with user-friendly tools). This step requires knowledge of all input data location and delivery method. This includes information regarding any other applications that are prerequisites (or must be run simultaneously) with the simulation. The reference to prerequisite applications must be described in the simulation's MDL. This step also requires knowledge of which parameters are applicable to virtual test and what are the reasonable boundaries for them.
2. Preparation for Executing Parametric Trials.	Simulations differ in how they are initialized. The VPG must incorporate a capability to change simulation parameters. (Currently, this is achieved by explicitly citing parameters and their initial values in the MDL.) Some simulations allow (or require) alternatives to specifying fixed parameters values on a command line. They may require launching an interface GUI specific to that simulation. (In reality, this could be something as simple as running an editor to change the input data file[s].) However, there are configuration considerations in doing this. The tester would want to (and the VPG environment ought to) maintain a record of what was run, when it was run, and what were the initial conditions. Keeping track of such information will prove difficult if the VPG cannot ascertain the initial condition set by a simulation's interface GUI.



Steps to building and executing scenarios

Recall the steps needed to run scenarios:



The following are links to example cgi scripts and forms that sub-out to executables (which call vpg api routines).

VPG HTML Server Examples:

- Build a Scenario Model to see an example of a converting a Tool into a Scenario Model.
- Build a Scenario to see an example of a scenario build
- Execute a Scenario to see an example of an execute Scenario form.
- List Tools listing of saved Tools.
- List Scenario Models listing of saved Tools.
- List Scenarios listing of saved Tools. (Or if you know the command to give the server, you can accomplish the same thing with a POST FORM .
- Delete Tools a TOOL delete FORM.

Figure 12. Scenario Build and Execution Steps.

Note that the dashed cell entitled "Define & Load Tools" has already occurred. "Defining Tools" is the manual process referred to in the previous subsection (3.2.2.1). "Loading tools" can take place by starting the VPG database and interactively loading MDL files just defined. Alternatively, one could initiate the VPG main GUI (discussed in subsection 3.1.2.2.1 VPG GUI) to load these MDLs, but (currently) initiating the VPG main GUI is also a manual process. In either event, the defining and loading tools must occur before launching the scenario build and execution phase (via external web-based tools shown in Figure 12).

The next step toward creating a test scenario is to define an instance of a tool for use in the scenario. This instance (called a scenario model) was described in Table 3. Creating a scenario model is started by selecting the underlined "Build a Scenario Model" hypertext seen in Figure 12. Doing so reveals the menu shown in Figure 13.

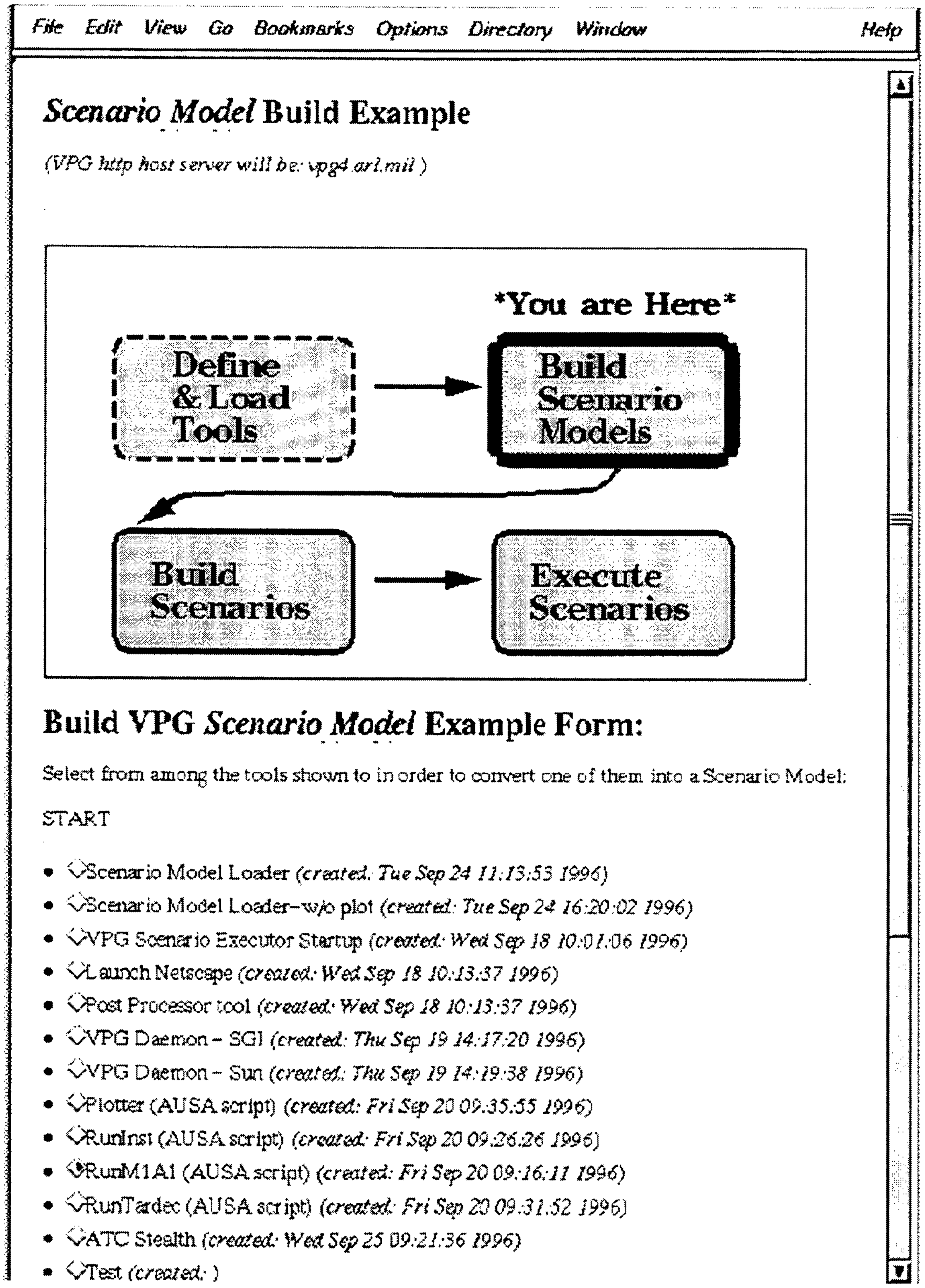
This menu presents tools found in the VPG database. Choosing any one of them and then selecting the "submit" button (which is scrolled off the bottom of the screen) will create a scenario model from the chosen tool. (Notice the test designer has chosen to create a scenario model from the "RunM1A1 [Association of the U.S. Army {AUSA} Script]" tool.)

After creating scenario models from a number of tools, it is time to combine those models into a logical test case scenario. Selecting the underlined "Build a Scenario" hypertext from Figure 12 presents the menu shown in Figure 14.

Building a scenario consists of indicating which scenario model(s) will participate. In this instance, all scenario models shown (two) have been selected. The scenario is created when the "submit choices" button is selected. Note that the test directory has named this scenario "Trial#5 Vehicle 5m/s CrossWind 0." ("CrossWind 0" has scrolled off the visible portion of the "Enter scenario name:" field.)

With the scenario created, the last step is to execute it. Figure 15 shows the execute scenario display. (This display is activated when the underlined "Execute a Scenario" hypertext is chosen from the menu shown in Figure 12.)

Using this menu, the VPG tester can run or re-run scenarios by selecting and activating the "submit choices" button. (Notice that "Trial#5 Vehicle 5m/s CrossWind 0" has been designated for execution.)



Scenario Model Build Example

(VPG http host server will be: vpg4.arl.mil)

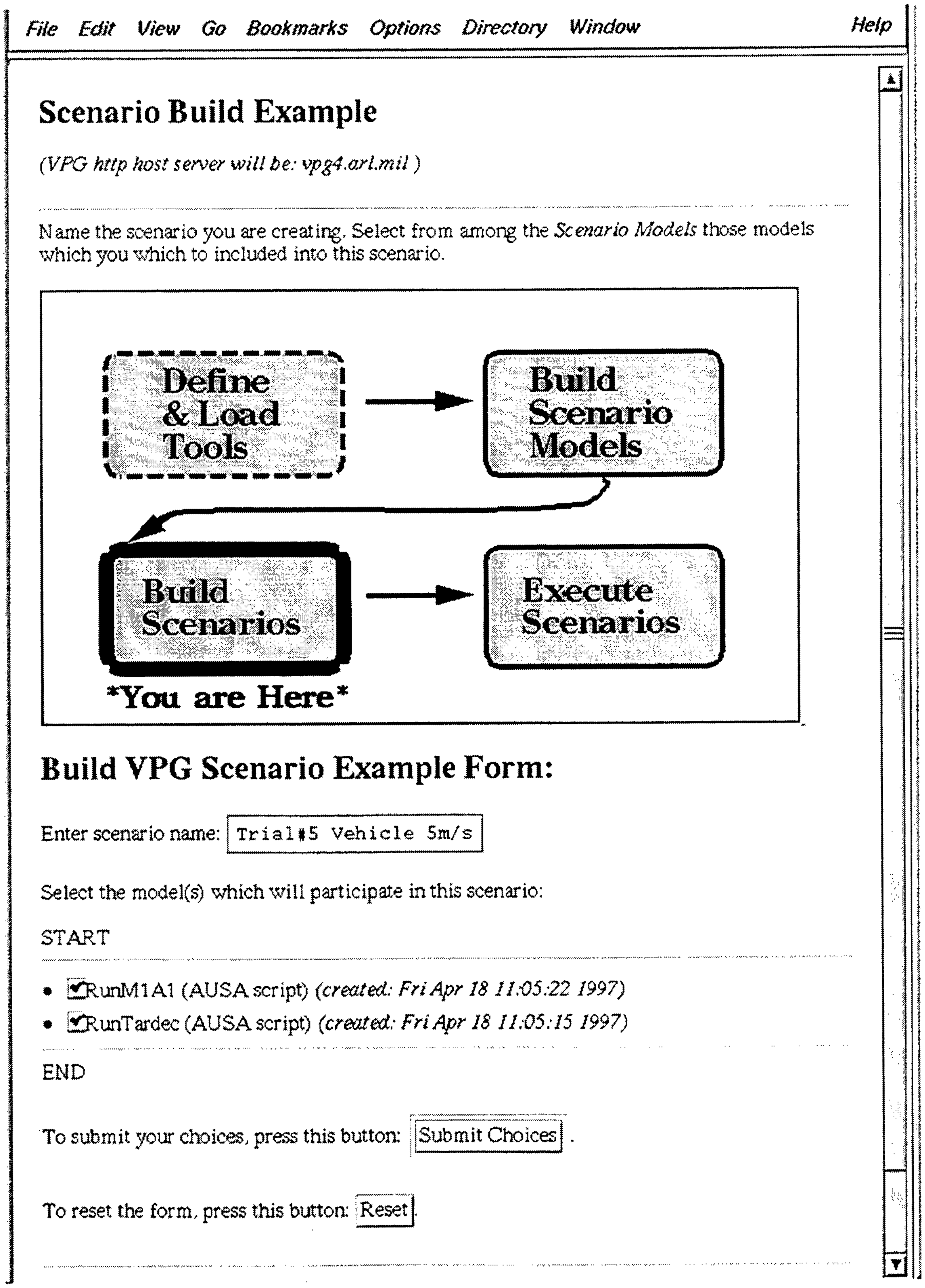
Build VPG *Scenario Model* Example Form:

Select from among the tools shown to in order to convert one of them into a Scenario Model:

START

- ◇Scenario Model Loader *(created: Tue Sep 24 11:13:53 1996)*
- ◇Scenario Model Loader--w/o plot *(created: Tue Sep 24 16:20:02 1996)*
- ◇VPG Scenario Executor Startup *(created: Wed Sep 18 10:01:06 1996)*
- ◇Launch Netscape *(created: Wed Sep 18 10:13:37 1996)*
- ◇Post Processor tool *(created: Wed Sep 18 10:13:37 1996)*
- ◇VPG Daemon - SGI *(created: Thu Sep 19 14:17:20 1996)*
- ◇VPG Daemon - Sun *(created: Thu Sep 19 14:19:38 1996)*
- ◇Plotter (AUSA script) *(created: Fri Sep 20 09:35:55 1996)*
- ◇RunInst (AUSA script) *(created: Fri Sep 20 09:26:26 1996)*
- ◇RunM1A1 (AUSA script) *(created: Fri Sep 20 09:16:11 1996)*
- ◇RunTardec (AUSA script) *(created: Fri Sep 20 09:31:52 1996)*
- ◇ATC Stealth *(created: Wed Sep 25 09:21:36 1996)*
- ◇Test *(created:)*

Figure 13. Scenario Model Build Example.



Scenario Build Example

(VPG http host server will be: vpg4.arl.mil)

Name the scenario you are creating. Select from among the *Scenario Models* those models which you which to included into this scenario.

Build VPG Scenario Example Form:

Enter scenario name: Trial#5 Vehicle 5m/s

Select the model(s) which will participate in this scenario:

START

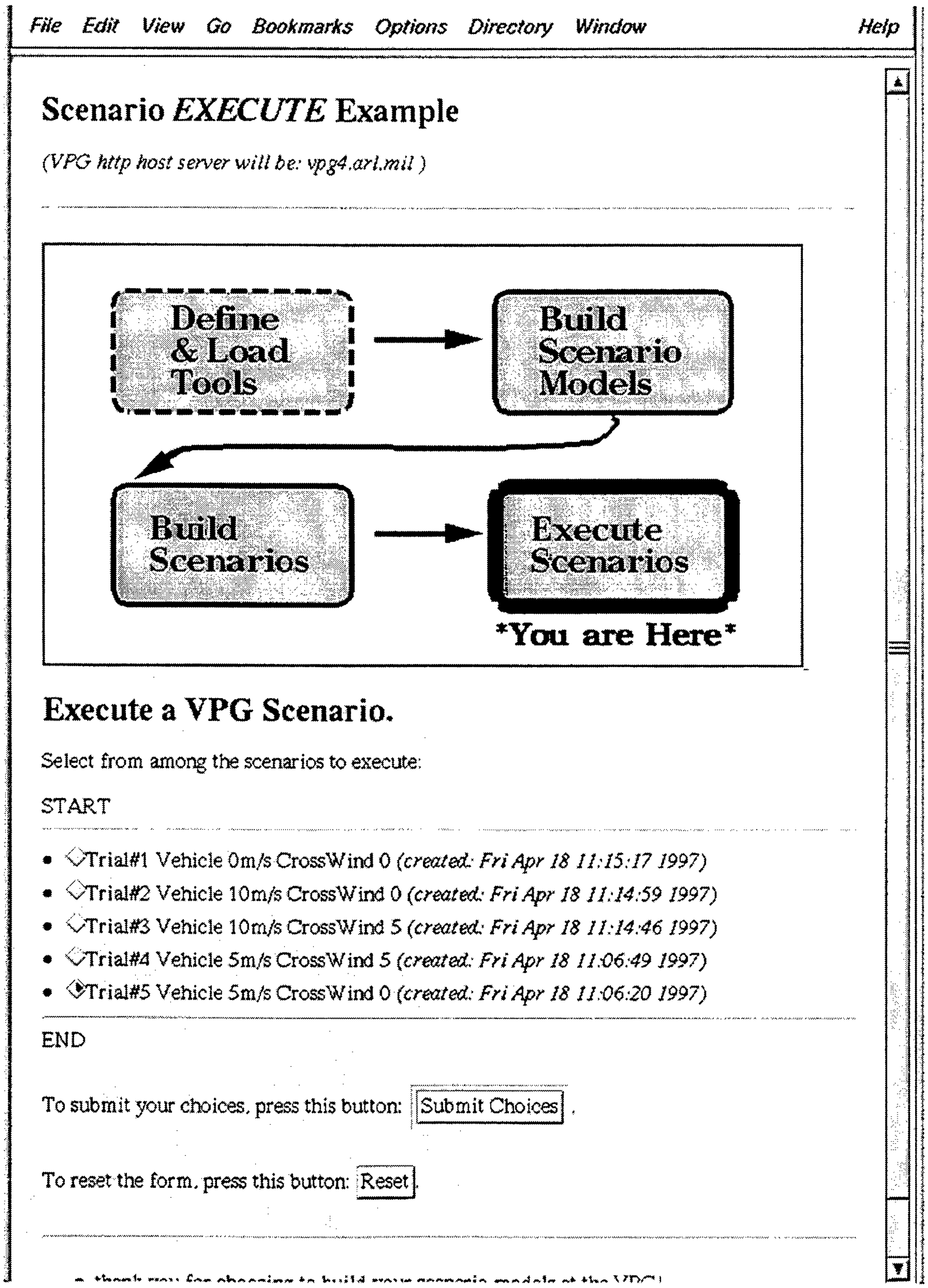
- ✓RunM1A1 (AUSA script) *(created: Fri Apr 18 11:05:22 1997)*
- ✓RunTardec (AUSA script) *(created: Fri Apr 18 11:05:15 1997)*

END

To submit your choices, press this button: Submit Choices .

To reset the form, press this button: Reset.

Figure 14. Scenario Build Example.



Scenario *EXECUTE* Example

(VPG http host server will be: vpg4.arl.mil)

Execute a VPG Scenario.

Select from among the scenarios to execute:

START

- ◇Trial#1 Vehicle 0m/s CrossWind 0 *(created: Fri Apr 18 11:15:17 1997)*
- ◇Trial#2 Vehicle 10m/s CrossWind 0 *(created: Fri Apr 18 11:14:59 1997)*
- ◇Trial#3 Vehicle 10m/s CrossWind 5 *(created: Fri Apr 18 11:14:46 1997)*
- ◇Trial#4 Vehicle 5m/s CrossWind 5 *(created: Fri Apr 18 11:06:49 1997)*
- [✓ Trial#5 Vehicle 5m/s CrossWind 0] *(created: Fri Apr 18 11:06:20 1997)*

END

To submit your choices, press this button: Submit Choices .

To reset the form, press this button: Reset.

Figure 15. Scenario Execute Example.

3.2.2.3 Post Processing and Analysis (a manually intensive task)

The purpose of the post processing and analysis component of Project Focus was to explore the problem of integrating analytical and display tools in the virtual test process. These tools are vital because they transform the raw test results into a form much more easily understood by the test customers, engineers, and program manager. Being able to easily post process (examine, filter, analyze) results of simulations is extremely crucial to a virtual testing environment. In fact, the whole point of running a model is being able to examine the results.

In an ideal simulation environment, users would be able to identify data variables of interest within a simulation and then specify how and when they were displayed or represented (simple enough). However, simulations developed independently are almost always going to describe their results (and required inputs) in different ways. Tables 5 and 6 are good examples. Both describe the same semantic content (vehicle path information), but they appear in very different forms. Even so, almost surprisingly, they have many similarities. One of them could have easily been a binary file (instead of ASCII), used a different coordinate system, or a multitude of other variations. This presents a complicated translation dilemma for a generalized post processing service. Yet such translations ought to be possible in some manner (and as transparent to the user as is reasonable). To conduct any translation, one must know two very specific things concerning the data in question: (a) their semantics (what do the data say?) and (b) their syntax (how do they say it?). Each data set (model inputs and outputs) would have to be described with this "data description language." In fact, such a language was being developed during Project Focus, the MDL discussed in Sections 3.1 Overall Architectural Structure and 3.1.1 The Architectural Core. Unfortunately, the role of the MDL would have to be greatly expanded to conduct translations "on the fly." This would require a highly complex descriptive language, data dictionary, and language parser. Once this is done, we still have not solved the original problem (implementing an integrated post processing service). Finishing the job would require another VPG library and tools supported by that library. The library would provide services such as parsing though output files (of different formats) or monitoring network traffic (looking for specific data items). These data items would then be collected into a logger file so that they would be available for further filtering or analysis. Supporting tools would allow the tester to control these actions through a user-friendly and intuitive interface. However, the implementation of such a complete capability was beyond the scope of Project Focus.

As a compromise, in order to have something implemented in time for Project Focus, it was decided to imitate most of this functionality. A simple script-based facility was implemented that allowed the user to specify what data from the simulation were to be processed and how. The user specified in plain English text what display program was needed for post processing, what data set was to be processed, and how the data set was to be processed. This partial solution required the user to be far more familiar with the internal nature of the post processing tools than was desired. MDL files were then constructed which provided the information need by the VPG core to find the outputs and launch the post process application. (In this case, the user had chosen a 2D/3D data plotting application called GNUplot.) Finally, the VPG main GUI's "post process data analysis" button (see Figure 5) was used to query for this MDL description and then launch the post process control application. The results were post process visual plots of certain critical outputs from ATC, ARL, and TARDEC simulations (shown in Figures 16 and 17).

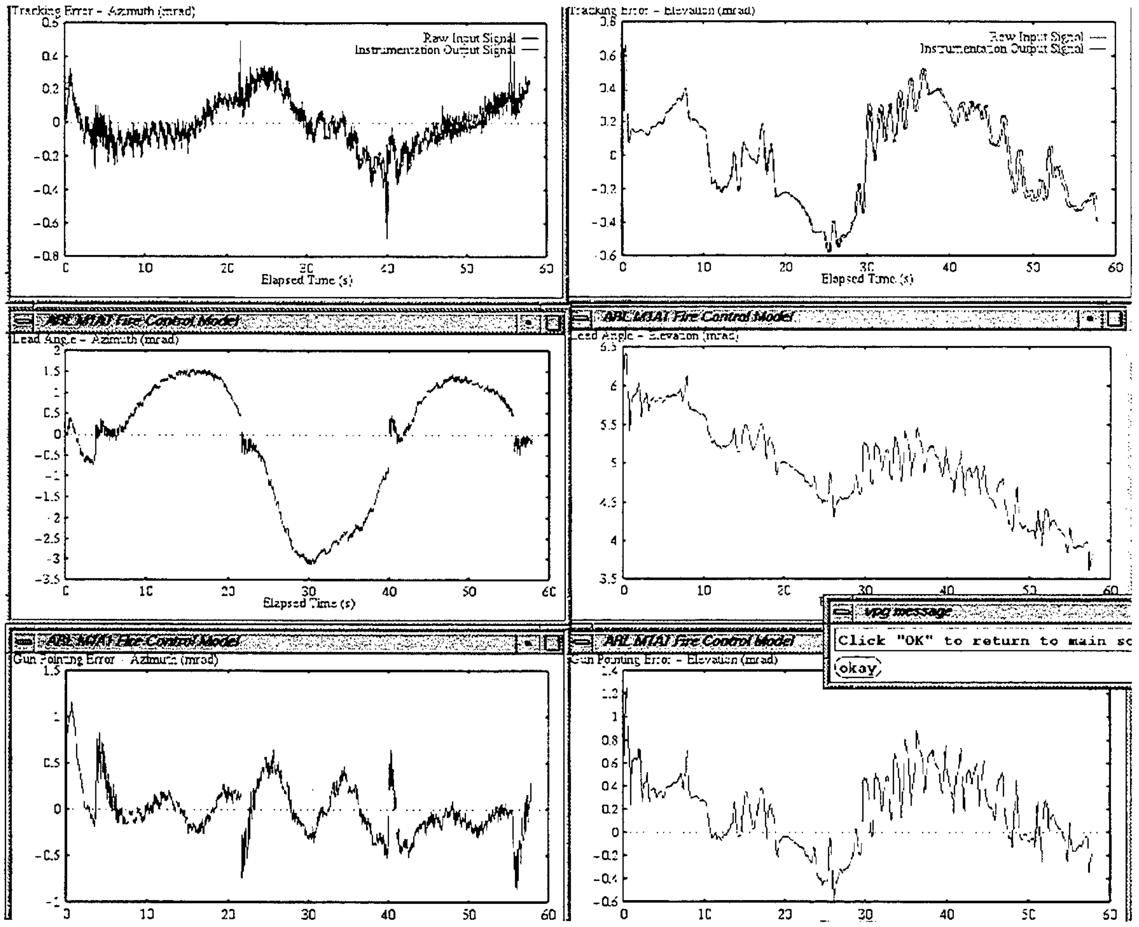

Tracking Error - Azimuth (mrad)
Tracking Error - Elevation (mrad)
Raw Input Signal
Instrumentation Output Signal
Elapsed Time (s)
ARL M1A1 Fire Control Model
Lead Angle - Azimuth (mrad)
Lead Angle - Elevation (mrad)
Gun Pointing Error - Azimuth (mrad)
Gun Pointing Error - Elevation (mrad)
vpg message
Click "OK" to return to main sc
okay

Figure 16. Gun Pointing Errors and Other Measurements From the ARL and ATC Models.

Although the post processing worked flawlessly, Project Focus demonstrated the need for a complete and integrated approach to post processing. A standard data journalizing tool and a robust set of post processing tools should be available to the user. These tools should minimize the worry and hassle of data management and let the user concentrate on the business of the simulation.

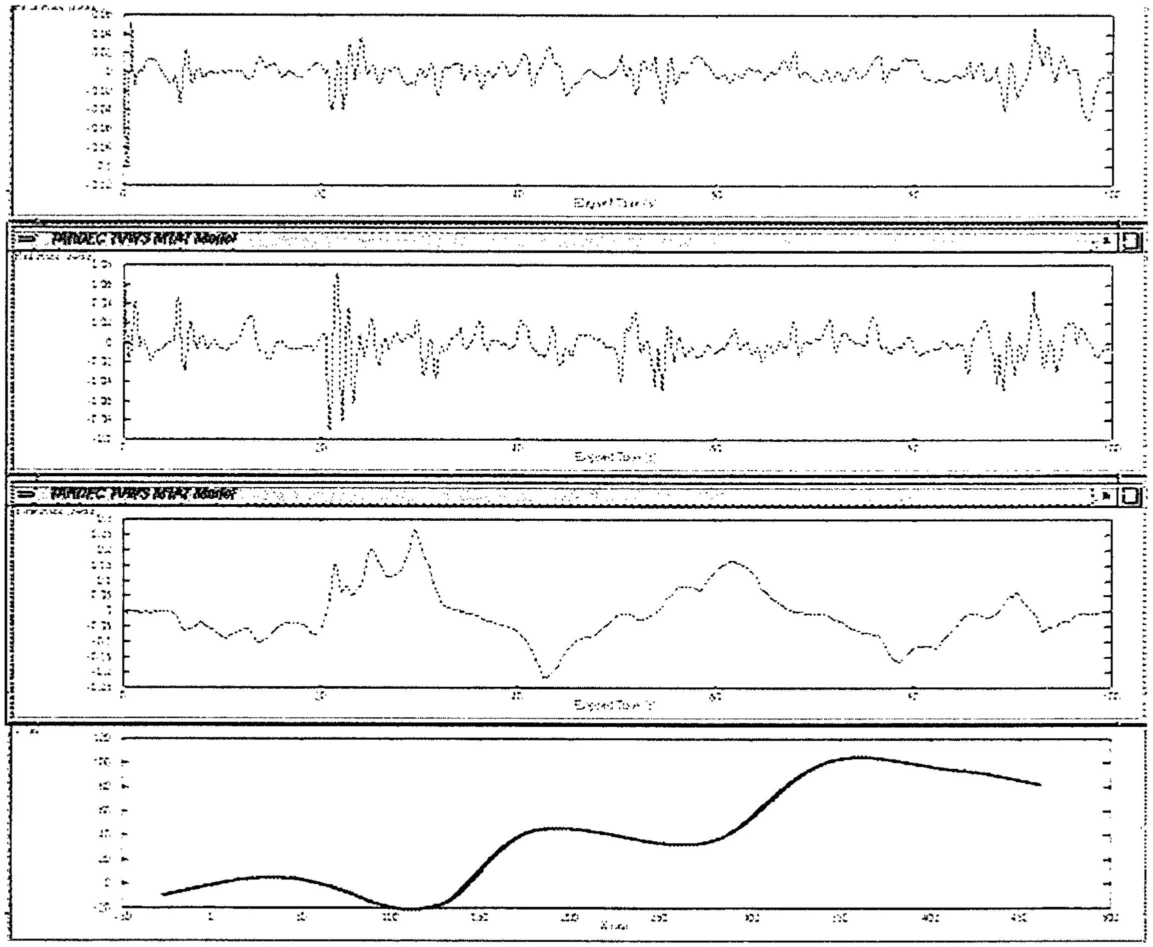


Figure 17. Pitch, Roll, Yaw, and Other Measurements From the TARDEC Model.

4. ARCHITECTURE REQUIREMENTS

Before Project Focus, a list of preliminary requirements was generated. It was supposed that these architectural requirements would satisfy the tester's needs. One of Project Focus's objectives was to inspect these requirements in the light of having conducted a trial case scenario using a prototype architecture. Following Project Focus, the original requirements were reviewed. This section presents the results of this review process. For reference and historical reasons, the original requirements are presented in their virgin state in Appendix D.

The intent of a requirements list is to provide a fundamental checklist that can be used as a basis to examine the qualities of a product (which looks promising to achieve the vision for the VPG).

The requirements list is divided into two levels: (a) a high functional level and (b) a lower, technical level.

We believe these help define an environment that allows modeling and simulation software components (loosely, models and data) to be organized in a consistent and cooperative framework (architecture) that allows effective communications, interoperability, and interchangeability among components.

4.1. Critical Functional Requirements

4.1.1. *Testing and Training*

The architecture shall provide seamless and user-oriented ways and means to design, implement, and conduct engineering level testing and constructive force-on-force engagements for operational and training exercises using modeling and simulations. This is the primary functional requirement for the VPG interface architecture. All other architectural requirements in some way or another relate to this requirement.

4.1.2. *Interoperability*

The architecture shall provide ways and means to achieve interoperability among models and simulations.

4.1.2.1. Interchangeability

The architecture shall incorporate interfaces to provide for plug-and-play software and hardware for models and simulations and their components.

4.1.2.2. Simulation Controls

The architecture shall provide simulation controls to coordinate and orchestrate models and simulations into multiple simulation exercises or a larger single simulation.

4.1.2.3. Access

The architecture shall provide interfaces to facilitate access to models and simulations.

4.1.2.4. Incorporate Models

The architecture shall provide ways and means to incorporate "disaffected" models and simulations. Disaffected models and simulations are legacy software and more generally, models developed outside the structure of the VPG interface architecture described herein. These are "alien" or unfriendly models.

4.1.3. *Re-use*

The architecture shall provide ways and means to reuse models and simulations and their components.

4.1.4. *Extendibility*

The architecture shall provide ways and means to extend the architecture to incorporate commercial off-the-shelf technologies (COTS) or Government-developed software tools.

4.2 <u>Critical Technical Requirements</u>

A. The architecture shall not be necessarily bound to a single machine. Users shall be able to remotely access architecture procedures and services from various platforms.

B. The architecture shall not be specific to one hardware system (machine). That is, the architecture should not be so tightly coupled to a particular vender's operating system and hardware that it cannot be ported to other operating systems of a similar nature.

C. The architecture shall not impede or prevent distributed simulations across hardware platforms. The architecture shall be capable of starting distributed portions of applications that are already distributed. Additionally, the architecture shall be capable of starting multiple simulations that are globally part of one exercise.

D. The architecture shall support load balancing. The architecture shall be capable of monitoring and recording resource use and performance parameters of simulation components during execution.

- The architecture shall be capable of arbitrating resource contention (i.e., with respect to models, data, etc.) by means of various methods to include first come, first-served; "round robin" time sharing; or a user-designated priority system.

E. The architecture shall allow simultaneous multiple simulations. The architecture shall be capable of maintaining control of multiple simulations executed concurrently by multiple users. These simulations may be stand-alone models or an integration of individual models running on different machines with independent or synchronized simulation clocks. The architecture shall have the ability to coordinate, execute, pause, suspend, monitor, save, resume, iterate, record, and replay distributed and non-distributed simulations.

F. The architecture shall provide time synchronization services. These services shall include real-time, non-real-time, and event-based time synchronization.

G. The architecture shall allow the inclusion or replacement of models and simulation components without having to rebuild the architecture core program.

H. The architecture shall provide automated simulation communication services. These services shall include the means for an individual user (i.e., simulation) to request a specific data transport or to independently manage its own data transport.

1. The architecture shall be capable of external communication by means of transmission control protocol/internet protocol (TCP/IP) protocols.

2. The architecture shall be capable of external communication by means of distributed interactive simulation (DIS) protocols.

3. The architecture shall be capable of external communication by means of RS232/RS422 serial ports.

4. The architecture shall support serial line internet/point-to-point protocol (SLIP/PPP) and common modem protocols.

I. The architecture shall incorporate an interface description service to allow simulation interfaces to be registered and documented with configuration and control mechanisms. The interface description service shall provide the means for applications to request interface descriptions of other applications to include the following data:

1. Input/output data descriptions to include data formats, data units, and data out-of-range bounds.

2. Hardware and software requirements needed to run the model.

3. Known sites where the model can be run and what connectivity prerequisites are needed.

4. Command line arguments that can be used to initiate execution.

5. Simulation time controls that are used by the model.

6. Metrics of fidelity that can be used to determine if the simulation is compatible.

7. Verification, validation, and accreditation (VV&A) certifications that have been applied to the model.

8. Documentation describing the physics of the model.

J. The architecture shall provide a transparent and seamless I/O interface to logically aggregate individual models and simulations into higher level components.

1. The architecture shall provide the end user with a display of the logical connectivity of the overall simulation under the control of the architecture.

2. The architecture shall provide means for the end user to configure the overall simulation (under the control of the architecture), as well as the components it encompasses.

K. The architecture shall provide on-demand, transparent terrain database services.

L. The architecture shall provide the means for configuration management of models and simulations and their related data elements.

M. Security. The architecture's function shall not be disabled by attaching point-to-point (hardware) security devices on the computer network.

N. Permissions. User authentication (by password or other acceptable means) shall be required. File system elements (files, data, directories, devices, etc.) shall be owned by a single user at any one time. That is, ownership of computer and file system elements shall never be ambiguous in this multi-user multi-processing environment. Users shall be able to add (or remove), read, and/or write permissions to any of their owned elements. Permission shall be at least designatable for themselves, selected groups of users, or all other users.

5. SUMMARY

Practical virtual testing requires an environment in which simulations, data, procedures, (as well as modeler and tester expertise and experience) can be easily merged. ARL and ATC jointly developed a set of "first cut" functional requirements for such an environment. Using this baseline, a prototype software architecture was designed and developed. During Project Focus, several models of varying fidelity were applied in an exercise to demonstrate the proof of principle of these architectural concepts.

Based on the results of Project Focus, the baseline requirements were reviewed and refined.

6. CONCLUSIONS

In the VPG vision, simulations, data, procedures, tester experience, simulation developer/ user expertise are merged into a cohesive and approachable system. Achieving this vision could be greatly advanced by paying particular attention to the architecture aspects referred to in this section.

• *A mature simulation databasing capability is needed.* This databasing capability must extend to both simulations, their abilities, required inputs, intended use, as well as to all means of simulation data (inputs and outputs) and configuration records.

In Section 3, we mentioned that by selecting a particular TOP, the tester was assured that simulations applicable to that TOP would be available (and simulations that may not be appropriate were hidden). This was both a benefit and a drawback since tools where not matured enough to be able to conduct the extensive database parsing required to determine whether a tool was suited for a particular purpose. The lesson here is that we need a more mature database as well as database development tools in order to conduct the types of sophisticated procedures envisioned for the VPG. Two examples are complicated meta-data queries and data mining. We shall define meta-data queries to be searching through the types of data available (not the data themselves). Data mining is a term used to described the concept of compiling a useful synopsis from very general unorganized voluminous data sources.

Another place where mature data management tools can greatly enhance VPG testing is in the form of data loggers. The post processing procedure conducted for Project Focus was an unsatisfactorily manual procedure. What is needed is the capability to identify and keep track of simulations' outputs (for use in later post process analysis). Furthermore, tools for conducting the analysis should be highly integrated with the VPG environment. In this way, these tools can best employ the data logs and be able to present analysis options to the tester (in a user-friendly and highly automated manner).

• *Establish a VPG users' needs process.* A process needs to be established whereby VPG needs are communicated to live/system test directors. The test director should be required to satisfy these needs to the best of his or her ability (subject to requirements of the item being tested).

The value of captured test data (for re-use in the VPG) can be severely compromised by ignoring modeler's requirements. One notable example is the form in which positional measurements are taken in the field. Historical live tests (especially vehicular) need to reference geodetic coordinates. This is because the VPG terrain is tied to an earth coordinate reference frame (universal transverse mercator [UTM] coordinate system, earth-centered coordinate system, or other). It is recommended that, when it makes sense, all future field tests correlate geometric field measurements to an earth-referenced coordinate system of some type.

Since it is anticipated that this shall not be the last recommended practice, we strongly suggest that a process be established to communicate simulators' known needs to live test designers. One possible means of documenting these recommendations is to reflect them in the set of TOPs. In particular, suggestions should be incorporated in a revised TOP 3-2-602 (Vehicular Gun Stabilization Systems).

Note that documenting modeler's needs and ensuring that they are met by data collectors will not guarantee that the VPG will be able to repeat the results of live tests. Too many variations in physical test conditions (for the historical measurements) and assumptions made in the algorithms and conditions (for the simulated results) often make side-by-side comparison impractical. For these reasons, we do not ever expect "exact" matches between the real and virtual worlds. However, directors of live tests should be made aware of procedures that can enhance the usefulness of test measurements. Live test results and other field data remain invaluable for confirming (validating) general trends and simulation results.

APPENDIX A

VPG DATABASE (DB) API PROCEDURES

INTENTIONALLY LEFT BLANK

VPG DATABASE (DB) API PROCEDURES

This appendix displays database level application program interface (API) procedure calls. The VPG core architecture prototype developed for Project Focus was implemented in the "C" American National Standards Institute (ANSI) programming language. Proper calling syntax for each procedure is shown in either ANSI C or traditional (Kernigan and Ritchie) prototype forms.

The underlying database engine used in the Project Focus implementation was the distributed fact base (DFB) database management system. One of the attractive features of DFB is that it is a distributed database. The VPG database should be distributed in some manner (in the spirit of the architectural requirements. However, it was unnecessary to use this feature during Project Focus. That is, during Project Focus only one (centralized) database server was serving clients at any one time. DFB is Department of Defense (DoD)-developed software, but just about any contemporary database engine could be used. In the following procedure descriptions, when it is noted ***"DFB specific interpretation:"***, this specifies that the comment that follows applies to the specific database engine used (in this case, ***DFB***).

1. db_open().

Connect to a DFB. 'num_tries' specifies the number of times to try to connect before returning a failure.

Returns -1 on failure.

Prototype Syntax:

```
int db_open(hostname,prog_name,num_tries)
char     *hostname,*prog_name;
int      num_tries;
```

2. db_close().

Command the DB to terminate. This routine should be used by programs that are terminating or otherwise breaking their connection to the DB.

Prototype Syntax:

```
void db_close()
```

3. db_is_connected().

This routine returns TRUE if connected to the DB, FALSE otherwise.

Prototype Syntax:

```
int db_is_connected()
```

4. db_get_DB_hostname().

This routine returns the name of the machine that the DB is on for this connection. It returns NULL is not connected to the DB.

NOTE: it is the calling routine's responsibility to free the memory pointed to by the return.

Prototype Syntax:

```
char * db_get_DB_hostname()
```

5. db_obj_define().

This routine defines the structure of an object in the database. For the DFB this is equivalent to dkb_define(), where ther structure of a facttype is defined.

Returns: 0 on success, < 0 on failure.

Prototype Syntax:

```
int db_obj_define(facttype,args,num_args)
char    *facttype;              /* Type of fact being defined. */
DB_Arg  args[];
int     num_args;
```

6. db_obj_create().

This routine creates an object in the database. For the DFB, the object created is a fact of type "facttype". db_obj_create() is responsible for creating the fact in the DB and returning a pointer to the DB_Object that references it.

Prototype Syntax:

```
DB_Object       * db_obj_create(facttype,args,num_args)
char    *facttype;              /* Facttype of DFB fact to create.
*/
DB_Arg          args[];
int             num_args;
```

7. db_obj_remove().

This routine removes an object from the database that is identified via 'obj'.

Prototype Syntax:

```
void db_obj_remove(obj)
DB_Object       *obj;           /* Object to remove. */
```

8. db_obj_update().

Each DB_Arg contains the name of a field, the new value for the field, and the data type of the field in a particular database record. db_obj_update() then takes the DB_Arg value pairs identified by "args" and updates the data for each pair in the database object "db_obj".

DFB specific interpretation: (That is depending on what database engine is running under the API, this procedure may behave differently).

Returns 0 on success, -1 otherwise.

Prototype Syntax:

```
int db_obj_update(args,num_args,db_obj)
DB_Arg          args[];
int             num_args;
```

```
DB_Object        *db_obj;
```

9. db_obj_query().

Returns a list of database object identifiers. A database object is a fact or record. A database object identifier is the "handle" to that object so that it may be referenced, or accessed, later. 'num' is set to the number of objects that are in the list.

Note that it is the calling routines responsibility to free the DB_Object that is returned.

DFB specific interpretation:

This routine queries the DFB for all facts of type "obj_type" using the query string "query". It returns a linked list of DB_Objects which contain the factid of each matching fact. "num" is set to the number of items in the list, i.e., the number of facts that satisfied the query.

Prototype Syntax:

```
DB_Object    * db_obj_query(obj_type,query,num)
char     *obj_type,*query;
int      *num;
```

10. db_obj_set_arg_retrieve().

Sets up the DB_Arg value pair associating the "name" of a database object field with a memory location "data" of where to retrieve the value associated with "name". "type" indicates what type data "name" represents.

DFB specific interpretation:

This routine sets up a DB_Arg structure for subsequent use by db_obj_get_args() to perform a dkb_getfact and dkb_getvar to obtain the data for the fact field called "name".

Prototype Syntax:

```
void db_obj_set_arg_retrieve(DB_Arg *arg, char *name, void *data,
int type)
```

11. db_obj_set_arg_define().

Sets up the DB_Arg value pair to set the "name" of a database object field. "type" indicates what type data "name" represents. This is used inconjunction with db_obj_define().

DFB specific interpretation:

This routine sets up a DB_Arg structure for subsequent use by db_obj_define() to create a facttype definition.

Prototype Syntax:

```
void db_obj_set_arg_define(DB_Arg *arg, char *name, int type)
```

12. db_obj_set_arg_store().

Sets up the DB_Arg value pair associating the "name" of a database object field with a memory location "data" of where to store the value associated with "name". "type" indicates what type data "name" represents.

DFB specific interpretation:

This routine sets up a DB_Arg structure for subsequent use by db_obj_get_args() to perform a dkb_getfact and dkb_getvar to obtain the data for the fact field called "name".

Prototype Syntax:

```
void db_obj_set_arg_store(DB_Arg *arg, char *name, void *data, int
type)
```

13. db_obj_retrieve_args().

Takes the DB_Arg value pairs identified by "args" and retrieves the data for each pair from the database object "db_obj". Returns 0 if all value pairs were successfully retrieved, -1 if unable to access "db_obj", and a bit mask reflecting the value pairs that could not be retrieved if all value pairs were not retrievable. If no args are being retrieved return -2. Expects the 'value' field in the DB_Arg structure to be a pointer to where to store the data.

DFB specific interpretation:

This routine performs a dkb_getfact on "db_obj" and a dkb_getvar for each DB_Arg. Each DB_Arg contains the name of a field, the data type of the field, and a memory location of where to store the value of the field that is gotten from the fact pointed to by "db_obj". If the data type is a DF_STRING or DF_NAMREF, memory will be malloc'ed and value will be set to point to it. It is the calling routine's responsibility to free the memory.

Prototype Syntax:

```
long db_obj_retrieve_args(args,num_args,db_obj)
DB_ArgList      args;
int             num_args;
DB_Object       *db_obj;
```

14. db_obj_retrieve_list().

This routine takes a DB_Object and list name and returns a pointer to a DB_List linked list containing the data values of the elements found in "list". On failure, for any reason, a NULL pointer is returned. 'num' is set to the number of items in the list being returned, or -1 upon failure (for an empty list, 'num' is 0).

It is the calling routine's responsibility to free the list - the routine db_freeDB_List() exists for this purpose.

Prototype Syntax:

```
DB_List * db_obj_retrieve_list(DB_Object *db_obj, char *listname,
int *num)
```

15. db_obj_list_to_str().

This routine takes a DB_List converts it into a string suitable for use in a DB_Arg for creating or updating objects. A pointer to the string is returned. The calling routine is responsible for free'ing the returned string.

Prototype Syntax:

```
char * db_obj_list_to_str(DB_List *list)
```

16. db_copy_obj().

This routine copies the CONTENTS of one DB_Object, pointed to by "from_obj", into another DB_Object, "to_obj". It returns -1 if either DB_Object is NULL; returns 0 otherwise.

Prototype Syntax:

```
int db_copy_obj(to_obj, from_obj)
DB_Object       *to_obj, *from_obj;
```

17. db_dup_obj().

This routine duplicates the orig_obj and returns a pointer to the new copy of it.

Prototype Syntax:

```
DB_Object *db_dup_obj(orig_obj)
DB_Object *orig_obj;
```

18. db_obj_cmp_id().

This routine compares two DB_Objects to determine if they are the same. It is similar to bcmp(), but bcmp() cannot be used because DB_Object is a structure that contains a "next" field and it is not necessary that the "next" fields in two DB_Objects be identical. If either of the two objects are NULL, returns -1.

Returns 0 if identical, non-zero otherwise.

Prototype Syntax:

```
int db_obj_cmp_id(obj1,obj2)
DB_Object        *obj1,*obj2;
```

19. db_print().

Prints the message and the database object identifier.

DFB specific interpretation:

This routine prints the factid in "db_obj".

Prototype Syntax:

```
void db_print(msg,db_obj)
char             *msg;
DB_Object        *db_obj;
```

20. db_objid_to_str().

This routine takes a DB_Object's obj_id and returns a pointer to an ASCII string representation of it. Note that it is the calling routine's re- sponsibility to free the return string. If called with a NULL DB_Object for 'db_obj' this routine returns a string for the NULL_DB_Object.

Prototype Syntax:

```
char     * db_objid_to_str(db_obj)
DB_Object        *db_obj;
```

21. db_str_to_objid().

This routine takes an ASCII string representation of a DB_Object obj_id and returns a pointer to a DB_Object that contains the converted DB_Object obj_id. Note that it is the calling routine's responsibility to free the memory pointed to by the return.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static  DB_Object      * db_str_to_objid(db_obj_str)
char    *db_obj_str;
```

22. NULL_DB_Object().

This routine returns a pointer to a DB_Object whose 'obj_id' is 0. This represents a NULL DB_Object.

Prototype Syntax:

```
DB_Object *NULL_DB_Object()
```

23. dbl_fid_to_obj().

This routine turns a DFB factid into a DB_Object. It is similar to db_str_to_objid(), except that db_str_to_objid() takes an ASCII version of 'fact_id' not a DFB representation.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static DB_Object *dbl_fid_to_obj(fact_id)
dkb_factid_t    fact_id;
```

24. db_freeDB_List().

This routine frees a linked list of DB_List structures.

Prototype Syntax:

```
void db_freeDB_List( DB_List *list )
```

25. db_free_DB_List_struct().

This routine frees the memory associate with a DB_List structure. It assumes the struct was malloc'ed to begin with and the 'data' field references malloc'ed memory as well if the type is DB_STRING, DB_LIST, or DB_OBJECT.

Prototype Syntax:

```
void db_free_DB_List_struct( DB_List *list )
```

26. db_free_DB_Object_struct().

This routine frees the memory associated with a DB_Object structure. It assumes the DB_Object was malloc'ed to begin with.

Prototype Syntax:

```
void db_free_DB_Object_struct(DB_Object *obj)
```

27. db_obj_list_append_item().

Append an item, "newItem", to a list, "listName", in the fact, "pObj".

Returns 0 on success, -1 on failure.

Prototype Syntax:

```
int db_obj_list_append_item(DB_Object *pObj, char *listName, void
*newItem, int type)
```

28. db_notify_check().

This routine checks to see if the DB has sent notice of changes to information that is of interest to the calling program. Such interest is registered with the DB via the db_notify() routine.

Note: in the DFB notices are referred to as triggers.

Returns 1 if a notice is pointed to by ret_notice, 0 if no notices, -1 if not connected to DB, and -2 if an error of some type is detected. It is the calling routines responsibility to free the notice_q_item structure pointed to by 'ret_notice'.

Prototype Syntax:

```
int db_notify_check(ret_notice)
struct  notice_q_item   **ret_notice;
```

29. db_notify_set().

This routine enables the calling application program to register with the DB "interest" that it has in changes to a particular type of record (fact) or more specifically certain fields within a record (fact).

Note: for the DFB implementation of the DB this is referred to as setting triggers.

Returns: -1 on failure, 1 on success.

Prototype Syntax:

```
int db_notify_set(handle, obj_type, cond)
char    *handle;        /* String by which triggers are
identified. */
char    *obj_type;      /* Object type of interest. */
char    *cond;          /* Conditions that specify interest in
obj_type. */
```

30. db_notify_cancel().

This routine is for cancelling previous notification (trigger) requests. Notifications are identified by their 'handle'.

Note: for the DFB implementation of the DB notifications are called triggers.

Prototype Syntax:

```
void db_notify_cancel(handle)
```

31. notice_handler().

Called when triggers are pending on the DFB pkg connection. Triggers are queued for later processing. The are de-queued in db_notify_check(). This is to try to avoid a messy situation where a trigger might arrive (asynchronously) while this routine is blocking awaiting the arrival of a synchronous DFB response - i.e., this routine issued a DFB command. The problem is that an incoming trigger would be misinterpreted as the expected DFB response.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static void notice_handler(pc,buf,length)
struct pkg_conn *pc;
int             length;
char            *buf;
```

32. dbl_pkg_error().

Responds to MSG_ERROR or MSG_SYNREQ pkg message.

Prototype Syntax:

```
void dbl_pkg_error(type,buf,len)
int     type,len;
char    *buf;
```

33. dbl_do_synch_resp().

Complains about an unexpected DFB synchronous response.

Prototype Syntax:

```
void dbl_do_synch_resp(pc,buf)
struct  pkg_conn *pc;
char             *buf;
```

INTENTIONALLY LEFT BLANK

APPENDIX B

VPG API PROCEDURES

INTENTIONALLY LEFT BLANK

VPG API PROCEDURES

This section lists most of the general VPG API routines in the VPG core discussed in Section ***3.1.1.2 VPG Core: VPG API***. The VPG core architecture prototype developed for Project Focus was implemented in the "C" (ANSI) programming language. Proper calling syntax for each procedure is shown in either ANSI C or traditional (Kernigan and Ritchie) prototype forms. This appendix is not meant to be a tutorial; its purpose is to display the prototype API in order to present a flavor for this VPG layer.

The naming convention used was to prologue VPG API system calls with "**vpg_**" followed by an acronym alluding to the object or function serviced by that library. (For example, all software functions manipulating the "tool" object would be called named "**vpg_tool_*something***" (e.g., vpg_tool_create(), vpg_tool_destroy(), etc.)

1. vpg_link_new().

The VPG_LINK and VPG_LLIST libraries are a specialized adaptation of a general linked list. It can be used for handling any data objects internally.

This routine creates a new vpg_Link. 'pobj' must be provided as input.

Prototype Syntax:

```
vpg_Link * vpg_link_new(void *pobj)
```

2. vpg_link_free().

Delete a vpg_Link pointed to by plink. This function is also responsible for freeing the object it contains. The function to free Obj, (obj_free) is provided as input.

Prototype Syntax:

```
void vpg_link_free(vpg_Link *plink, void (*obj_free)(void *pobj))
```

3. vpg_llist_new().

The VPG_LINK and VPG_LLIST libraries are a specialized adaptation of a genral linked list. It can be used for handling any data objects internally.

vpg_llist_new() creates a vpg_Llist structure and returns a pointer to it (or a NULL if unable to create it).

Prototype Syntax:

```
vpg_Llist *vpg_llist_new()
```

4. vpg_llist_free().

Delete a vpg_Llist pointed to by pllist. This function is also responsible for the deletion of all the vpg_links it contains and all the objects contained in vpg_Link's. pllist must be provided as input and the function for deleting an object, (*obj_free), should also be provided.

Prototype Syntax:

```
void vpg_llist_free(vpg_Llist *pllist, void (*obj_free)(void
*pobj))
```

5. vpg_llist_size().

Get the number of vpg_Link's in the vpg_Llist pointed to by pllist.

Prototype Syntax:

```
int vpg_llist_size(vpg_Llist *pllist)
```

6. vpg_llist_addhead().

Add a new vpg_Link in the vpg_Llist at the head

Prototype Syntax:

```
void vpg_llist_addhead(vpg_Llist *pllist, void *pobj)
```

7. vpg_llist_addtail().

Add a new vpg_Link in the vpg_Llist at the tail.

Prototype Syntax:

```
void vpg_llist_addtail(vpg_Llist *pllist, void *pobj)
```

8. vpg_llist_delete().

Delete a link from the llist.

Prototype Syntax:

```
void vpg_llist_delete(vpg_Llist *pllist, vpg_Link *plink,void
(*obj_free)(void *pobj))
```

9. vpg_llist_head().

set the current link pointer pcl to head phead.

Prototype Syntax:

```
void vpg_llist_head(vpg_Llist *pllist)
```

10. vpg_llist_tail().

Set the current link pointer pcl to tail ptail.

Prototype Syntax:

```
void vpg_llist_tail(vpg_Llist *pllist)
```

11. vpg_llist_istail().

Check if the current pointer pcl is pointing to the last link

Prototype Syntax:

```
int vpg_llist_istail(vpg_Llist *pllist)
```

12. vpg_llist_ishead().

check if the current pointer pcl is pointing to the first link

Prototype Syntax:

```
int vpg_llist_ishead(vpg_Llist *pllist)
```

13. vpg_llist_next().

advance the current link pointer pcl to point to the next link

Prototype Syntax:

```
void vpg_llist_next(vpg_Llist *pllist)
```

14. vpg_llist_previous().

Move the current link pointer pcl to point to the previous link

Prototype Syntax:

```
void vpg_llist_previous(vpg_Llist *pllist)
```

15. vpg_llist_dup().

Duplicate a llist.

Prototype Syntax:

```
vpg_Llist *vpg_llist_dup(vpg_Llist *pllist_orig, void
*(*obj_dup)(void *pobj))
```

16. vpg_llist_find().

Look for "lookfor" in the llist. If found, returns a ptr to the link that contains it; if not returns a NULL ptr.

note: an object match function must be provided.

Prototype Syntax:

```
vpg_Link *vpg_llist_find(vpg_Llist *pllist, void *lookfor, int
(*obj_match)(void *pobj, void *lookfor))
```

17. vpg_llist_search().

Search for "lookfor" in the llist. If found, returns 1, otherwise a 0 is returned.

Prototype Syntax:

```
int vpg_llist_search(vpg_Llist *pllist, vpg_Link *lookfor)
```

18. vpg_llist_print().

Prototype Syntax:

```
void vpg_llist_print(vpg_Llist *pllist, void (*obj_print)(void
*pobj))
```

19. vpg_mdl_getByName().

This routine returns a list of MDLs whose name is "tName". The name field should be unique, but that is not this routine's responsibility. If no is found with name "tName" then NULL is returned.

Returns: a linked list of vpgMDLs, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgMDL structs that are returned. (This will not release the instance of the vpgMDL in the DB.)

Prototype Syntax:

```
vpgMDL_t * vpg_mdl_getByName(char *tName)
```

20. vpg_mdl_getByKey().

This routine returns the MDL whose key is "tKey". The key field should be unique, but that is not this routine's responsibility. If none is found with key "tKey" then NULL is returned.

Returns: a pointer to a vpgMDL_t, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgMDL struct that are returned. (This will not release the instance of the vpgMDL in the DB.)

Prototype Syntax:

```
vpgMDL_t * vpg_mdl_getByKey(char *tKey)
```

21. vpg_mdl_getKey().

This routine returns the "key" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getKey(vpgMDL_t *mdl)
```

22. vpg_mdl_getName().

This routine returns the "MODEL_NAME" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getName(vpgMDL_t *mdl)
```

23. vpg_mdl_getExec().

This routine returns the "Executable" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getExec(vpgMDL_t *mdl)
```

24. vpg_mdl_getProgramPath().

This routine returns the "PROGRAM_PATH" field from a vpgMDL object. This field contains a value that looks like: "@hostname:path". There are separate routines for extracting just the "hostname" or the "path". This routine returns the entire value.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getProgramPath(vpgMDL_t *mdl)
```

25. vpg_mdl_getRunHost().

This routine returns the "Run Host" field from a vpgMDL object. The host is stored in the PROGRAM_PATH field which looks like: "@hostname:path". This routine just returns "hostname" from that string.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getRunHost(vpgMDL_t *mdl)
```

26. vpg_mdl_getExecPath().

This routine returns the "Executable Path" field from a vpgMDL object. The path is stored in the PROGRAM_PATH field which looks like: "@hostname:path". This routine just returns "path" from that string.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getExecPath(vpgMDL_t *mdl)
```

27. vpg_mdl_getFileType().

This routine returns the "File Type" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getFileType(vpgMDL_t *mdl)
```

28. vpg_mdl_getOutputInput().

This routine returns the "OUTPUT_INPUT" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getOutputInput(vpgMDL_t *mdl)
```

29. vpg_mdl_getGroups().

This routine returns the "Groups" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getGroups(vpgMDL_t *mdl)
```

30. vpg_mdl_getArgs().

This routine returns the "Arguments" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getArgs(vpgMDL_t *mdl)
```

31. vpg_mdl_getRunDir().

This routine returns the "Run Directory" field from a vpgMDL object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_mdl_getRunDir(vpgMDL_t *mdl)
```

32. vpg_mdl_setKey().

This routine sets the "key" field of an mdl to be "newKey".

mdl - mdl handle
newKey - new key for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setKey(vpgMDL_t *mdl, char *newKey)
```

33. vpg_mdl_setName().

This routine sets the "MODEL_NAME" field of an mdl to be "newName".

mdl - mdl handle
newName - new name for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setName(vpgMDL_t *mdl, char *newName)
```

34. vpg_mdl_setExec().

This routine sets the "PROGRAM_EXECUTABLE" field of an mdl to be "newExec".

mdl - mdl handle
newExec - new executable for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setExec(vpgMDL_t *mdl, char *newExec)
```

35. vpg_mdl_setProgramPath().

This routine sets the "PROGRAM_PATH" field of an mdl to be "newPath". This field contains a value that looks like: "@hostname:path". There are separate routines for setting just the "hostname" or the "path". This routine sets the entire value.

mdl - mdl handle
newPath - new program path for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setProgramPath(vpgMDL_t *mdl, char *newPath)
```

36. vpg_mdl_setRunHost().

This routine sets the "PROGRAM_PATH" field of an mdl to be "newPath_Host". "PROGRAM_PATH" is actually a field that contains a value that looks like: "@hostname:path". This routine sets only the "hostname" parth of the value.

mdl - mdl handle
newPath_Host - new program path hostname for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setRunHost(vpgMDL_t *mdl, char *newPath_Host)
```

37. vpg_mdl_setExecPath().

This routine sets the "PROGRAM_PATH" field of an mdl to be "new_path". "PROGRAM_PATH" is actually a field that contains a value that looks like: "@hostname:path". This routine sets only the "path" part of the value.

mdl - mdl handle
new_path - new program path for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setExecPath(vpgMDL_t *mdl, char *new_path)
```

38. vpg_mdl_setFileType().

This routine sets the "FILE_TYPE" field of an mdl to be "newFileType".

mdl - mdl handle
newFileType - new file type for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setFileType(vpgMDL_t *mdl, char *newFileType)
```

39. vpg_mdl_setOutputInput().

This routine sets the "OUTPUT_INPUT" field of an mdl to be "newOutputInput".

mdl - mdl handle
newOutputInput - new output_input for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setOutputInput(vpgMDL_t *mdl, char *newOutputInput)
```

40. vpg_mdl_setGroups().

This routine sets the "GROUPS" field of an mdl to be "newGroups".

mdl - mdl handle
newGroups - new groups for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setGroups(vpgMDL_t *mdl, char *newGroups)
```

41. vpg_mdl_setArgs().

This routine sets the "ARGUMENTS" field of an mdl to be "newArgs".

mdl - mdl handle
newArgs - new args for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setArgs(vpgMDL_t *mdl, char *newArgs)
```

42. vpg_mdl_setRunDir().

This routine sets the "RUN_DIR" field of an mdl to be "newRunDir".

mdl - mdl handle
newRunDir - new run directory for mdl

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_mdl_setRunDir(vpgMDL_t *mdl, char *newRunDir)
```

43. vpg_scenModel_create().

This call uses the tool (or MDL) pointed to by aModel is used as the basis for creating a "scenario model" fact in the database. The "key" field from "aModel" is used as the value for the "orig_model_r".

The resulting "scenario model" handle returned can then be used to include the tool as part of the tools run in a "scenario" (via the vpg_scen_addModel() call).

A unique key is generated (via vpgMakeKey()) and assigned to the scenModel's "key" field. All other fields are copied from the "aModel" object except for the "scenario_r" field which is not assigned until the scenModel is attached to a particular senario (via the vpg_scen_addModel() call).

Returns: handle to the new "scenario model" fact - success; NULL - failure.

Prototype Syntax:

```
vpgScenModel_t * vpg_scenModel_create( vpgMDL_t *aModel)
```

44. vpg_scenModel_destroy().

Deletes a "scenario model" fact from the DB. Any reference to it by its "scenario" fact (whose "key" is in "scenario_r") must be removed by the application programmer.

This is easily accomplished by first calling vpg_scenModel_getScenario_R() to get the scenario (if one exists) which uses this scenario model. Next a call to vpg_scen_RemoveModel() is made to remove the scenario Model from the scenario. Lastly vpg_scenModel_destroy() is called to delete the Scenario Model itself from the database.

Returns: 1 on success. 0 on failure.

Prototype Syntax:

```
int vpg_scenModel_destroy(vpgScenModel_t *aScenModel )
```

45. vpg_scenModel_getRunHost().

This routine returns the "Run Host" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getRunHost(vpgScenModel_t *scenModel)
```

46. vpg_scenModel_getRunName().

This routine returns the "run_name" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getRunName(vpgScenModel_t *scenModel)
```

47. vpg_scenModel_getExecPath().

This routine returns the "Executable Path" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getExecPath(vpgScenModel_t *scenModel)
```

48. vpg_scenModel_getExec().

This routine returns the "Executable" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getExec(vpgScenModel_t *scenModel)
```

49. vpg_scenModel_getArgs().

This routine returns the "Arguments" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getArgs(vpgScenModel_t *scenModel)
```

50. vpg_scenModel_getRunDir().

This routine returns the "Run Directory" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getRunDir(vpgScenModel_t *scenModel)
```

51. vpg_scenModel_getGroups().

This routine returns the "Groups" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getGroups(vpgScenModel_t *scenModel)
```

52. vpg_scenModel_getKey().

This routine returns the "key" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getKey(vpgScenModel_t *scenModel)
```

53. vpg_scenModel_getByKey().

This routine returns the scenModel whose key is "tKey". The key field should be unique, but that is not this routine's responsibility. If none is found with key "tKey" then NULL is returned.

Returns: a pointer to a vpgScenModel_t, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgscenModel struct that are returned. (This will not release the instance of the vpgscenModel in the DB.)

Prototype Syntax:

```
vpgScenModel_t * vpg_scenModel_getByKey(char *tKey)
```

54. vpg_scenModel_getScenario_R().

This routine returns the key of the scenario referenced by the "scenario_r" field of a scenario model.

Returns: a pointer to the key on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getScenario_R(vpgScenModel_t *scenModel)
```

55. vpg_scenModel_getOriginal_R().

This routine returns the key of the MDL (tool) referenced by the "orig_model_r" field of a scenario model.

Returns: a pointer to the key on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getOriginal_R(vpgScenModel_t *scenModel)
```

56. vpg_scenModel_getParseTree().

This routine returns the "parse_tree" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getParseTree(vpgScenModel_t *scenModel)
```

57. vpg_scenModel_getName().

This routine returns the "name" associated with a scenModel.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_scenModel_getName(vpgScenModel_t *scenModel)
```

58. vpg_scenModel_setScenario_R().

set the "scenario_r" field for the given scenModel_t to the character string pointed to by "key_str".

No checks are made as to the validity of the string being assigned to the field. It is up to the caller to ensure that it the "key" of a valid scenario in the database.

returns 1 on success. 0 on failure.

Prototype Syntax:

```
int vpg_scenModel_setScenario_R( vpgScenModel_t *sm,  char
*key_str  )
```

59. _vpg_scen_modelCreate().

Create/Destroy a "scenario_model" fact in the data base. These two functions are used internally in the API, so they are declared static.

modelKey - key to the model (same as the key to the corresponding mdl)

return values:
  for "vpg_scen_modelCreate()": fact handle - success; NULL - failure
  for vpg_scen_modelDestroy()": NONE

Note: for "vpg_scen_modelCreate()", the corresponding "mdl" fact has to be identified and accessed in order to retrieve info to fill out the "parse_tree", "run_host", "run_path" and "run_name" fields. Also note: it is the responsibility of the calling function to add the scenario Id to the fact an update the record in the database... (though this work break-down does not make sense).

Prototype Syntax:

NOTE: Not callable as an API - used internally in vpg_scenModel_create().

```
static vpgScenario_t *_vpg_scen_modelCreate( char *modelKey)
```

60. _vpg_scen_modelDestroy().

static void _vpg_scen_modelDestroy(char *modelKey);

Destroy the scenario_model. Remove it from the database and free allocated components within its structure. (Note: scenario_r is not affected by this function.)

Prototype Syntax:

NOTE: Not Callable as an API - used internally in vpg_scenModel_destroy().

```
static int _vpg_scen_modelDestroy( vpgScenModel_t *sm)
```

61. vpg_scen_create().

Create a scenario fact in the database. A unique key is generated (via vpgMakeKey()) and assigned to the "key" field. The "name" field is instantiated with "scenName". All other fields are uninitialized.

return value: handle to the "scenario" fact - success; NULL - failure.

Prototype Syntax:

```
vpgScenario_t * vpg_scen_create(char *scenName)
```

62. vpg_scen_destroy().

Deletes a "scenario" fact from the DB. All the "scenario_model" facts associated with the scenario must be deleted as well (using "vpg_scenModelDestroy()").

aScen - handle to the "scenario" fact to be deleted.

return value: NONE

Prototype Syntax:

```
void vpg_scen_destroy(vpgScenario_t *aScen)
```

63. vpg_scen_getScenarios().

This routine returns a list of scenarios from the DB. If none are found, then NULL is returned.

Returns: a linked list of vpgScenarios, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgScenario structs that are returned. (This will not release the instance of the vpgScenario in the DB.)

Prototype Syntax:

```
vpgScenario_t * vpg_scen_getScenarios()
```

64. vpg_scen_getByKey().

This routine returns the scenario whose key is "tKey". The key field should be unique, but that is not this routine's responsibility. If none is found with key "tKey" then NULL is returned.

Returns: a pointer to a vpgScenario_t, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgScenario_t struct that is returned. (This will not release the instance of the vpgScenario in the DB.)

Prototype Syntax:

```
vpgScenario_t * vpg_scen_getByKey(char *tKey)
```

65. vpg_scen_getByName().

This routine returns a list of scenarios whose name is "sName". The name field should be unique, but that is not this routine's responsibility. If none is found with name "sName" then NULL is returned.

Returns: a linked list of vpgScenarios, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgScenario structs that are returned. (This will not release the instance of the vpgScenario in the DB.)

Prototype Syntax:

```
vpgScenario_t * vpg_scen_getByName(char *sName)
```

66. vpg_scen_addModel().

Adds a scenario model to a scenario. Checks to see that scen_model is not already in the DB.

Associates the scenario model with this scenario (by having the "scenario_r" field of "scen_model" point to the scenario supplied (e.g. "aScen")).

Returns: 1 on success, 0 if scen_model is already in scenario, and -1 on failure.

Prototype Syntax:

```
int vpg_scen_addModel(vpgScenario_t *aScen,  vpgScenModel_t *
scen_model)
```

67. vpg_scen_removeModel().

Removes a model from a scenario. The model "aModel" is left untouched.

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_scen_removeModel(vpgScenario_t *aScen, vpgScenModel_t
*aModel)
```

68. vpg_scen_getName().

This routine retrieves the "name" from the given "scenario" fact.

Returns: a pointer (char *) to the name on success, NULL (char *)0 otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return.

Prototype Syntax:

```
char * vpg_scen_getName(vpgScenario_t *aScen)
```

69. vpg_scen_setName().

This routine sets the "name" field of a scenario to be "newName".

aScen - scenario handle
newName - new name for scenario

Returns: 1 on success, 0 on failure.

Prototype Syntax:

```
int vpg_scen_setName(vpgScenario_t *aScen, char *newName)
```

70. vpg_scen_getModels().

This routine retrieves the "models" from the given "scenario" fact. The return value is a pointer to a linked list of vpgScenModel_t structs.

Returns: a pointer to the first vpgScenModel_t in a linked list of vpgScenModels or NULL (if the list is empty).

Note: it is the calling routine's responsibility to free the vpgScenModel_t structs that are returned. This will not release the instance of the vpgScenModel_t in the DB. Use vpg_free_vpgScenModel_t().

Prototype Syntax:

```
vpgScenModel_t * vpg_scen_getModels(vpgScenario_t *aScen)
```

71. vpg_scen_getModelsKeys().

This routine retrieves the "models" from the given "scenario" fact. The return value is a pointer to a linked list of DB_List structures. The values in the structures are the keys of the models in the in the scenario. To access the actual value within the structure one would access the "s" union element of the "data" field. For example:

```
DB_List *list;
char    *key;

    list = vpg_scen_getModelsKeys(aScen);
    key = list->data.s;  /* list->type == DB_STRING */
```

Returns: a pointer to a linked list of DB_List structs on success, NULL on failure or if the list is empty.

Note: it is the calling routine's responsibility to free the DB_List structs that are returned. This will not release the instance(s) of the scenario models from the DB.

Prototype Syntax:

```
DB_List * vpg_scen_getModelsKeys(vpgScenario_t *aScen)
```

72. vpg_scen_getKey().

This routine retrieves the "key" from the given "scenario" fact.

Returns: a pointer (char *) to the key on success, NULL (char *)0 otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return.

Prototype Syntax:

```
char * vpg_scen_getKey(vpgScenario_t *aScen)
```

73. _vpg_scen_getModelsIds().

This routine retrieves the "model ids" from the given "scenario" fact. The return value is a pointer to a linked list of vpgScenModel_t structs.

Returns: a pointer to the first vpgScenModel_t in a linked list of vpgScenModels or NULL (if the list is empty).

Note: it is the calling routine's responsibility to free the vpgScenModel_t structs that are returned. This will not release the instance of the vpgScenModel_t in the DB. Use vpg_free_vpgScenModel_t().

Prototype Syntax:

```
static vpgScenModel_t *_vpg_scen_getModelsIds(vpgScenario_t
*aScen)
```

74. _vpg_scenGetModels().

DB_List *_vpg_scenGetModels(vpgScenario_t aSecn)

This routine retrieves the "models" from the given "scenario" fact. The Returns: a pointer to the list (represented by a linked list of DB_List structures) on success, or NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return. Use db_freeDB_List().

Prototype Syntax:

NOTE: Not callable as an API - used internally

```
static DB_List *_vpg_scenGetModels(vpgScenario_t *aScen)
```

75. vpg_sym_addSymbol().

This routine adds "symbols" and an associated value to the symbol table. The symbol table contains all variables that can be set through command line arguments, input files, programmatic defaults, or client inputs. The format for 'sym_exp' is: "SYMBOL_NAME value". New symbol_names are appended to the table, existing entries in the table are updated with the new value. 'src' identifies the source of the symbol (see vpg.h for values). If the state of a symbol value is STATIC the symbol value cannot be changed. If state is NONE, the state field is not changed. This allows a symbol's state to be set once when it is initialized. Subsequent calls to change a symbol's value do not need to know what state the symbol should be, they can simply set the state to NONE which leaves the original setting unchanged.

Returns: 1 on success, <1 otherwise.

Prototype Syntax:

```
int vpg_sym_addSymbol(char *sym_exp,  char *src,  int state)
```

76. vpg_sym_findSymbol().

This routine searches the DB symbol table for 'symbol'. If found, it returns a pointer to a symbol_identifer structure with the symbol info.

NOTE: It is the calling routine's responsibility to free the symbol_- identifier structure pointed to by the return.

Prototype Syntax:

```
struct  symbol_identifier * vpg_sym_findSymbol(char *symbol)
```

77. vpg_sym_printSymTable().

This routine prints the symbol table.

Prototype Syntax:

```
void vpg_sym_printSymTable()
```

78. vpg_sym_freeSymIdStruct().

This routine frees a malloc'ed symbol_identifier structure.

Prototype Syntax:

```
void vpg_sym_freeSymIdStruct(symid)

struct symbol_identifier *symid;
```

79. _vpg_sym_checkChangeImpact().

This routine checks for the impact of a changed parameter value. The effects of such a change could be far reaching. This routine is only called once it has been determined that a value will change, but BEFORE the change is made. So, symid is the old info and value, src and state is the new info. The following are the parameters whose change needs to be acted on:

Prototype Syntax:

```
void _vpg_sym_checkChangeImpact(symid,value,src,state)

struct  symbol_identifier       *symid;
char    *value, src;
int     state;
```

80. _vpg_sym_getDBObj().

This routine searches the DB for a symbol whose name matches 'sym_name'. It returns a pointer to the DB_Object that matches, or NULL.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static  DB_Object*_vpg_sym_getDBObj(sym_name)
char    *sym_name;
```

81. _vpg_sym_DBObjToSymId().

This routine converts a symbol DB object into a symbol_identifier structure.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static struct   symbol_identifier *_vpg_sym_DBObjToSymId(db_obj)

DB_Object       *db_obj;
```

82. _vpg_sym_getSymIdStruct().

This routine allocates memory (mallocs) and initializes a symbol_identifier structure. It returns a pointer to the new structure.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static struct symbol_identifier *_vpg_sym_getSymIdStruct()
```

83. vpg_tool_destroy().

remove the tool (or mdl) pointed to by "tool".

Returns
  1 on successful deletion.
  0 on failure.

Prototype Syntax:

```
int vpg_tool_destroy( vpgTool_t *tool )
```

84. vpg_tool_getToolList().

This routine returns a list of tools. In the VPG DB a tool is a program that can be started, or launched, by VPG that provides a service for VPG, but which is not compiled directly into the core VPG executable. Tools are actually just models.

Returns: a linked list of vpgTools, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgTool structs that are returned. (This will not release the instance of the vpgTool in the DB.)

Prototype Syntax:

```
vpgTool_t * vpg_tool_getToolList( )
```

85. vpg_tool_getByName().

This routine returns a list of tools whose name is "tName". The name field should be unique, but that is not this routine's responsibility. If no is found with name "tName" then NULL is returned.

Returns: a linked list of vpgTools, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgTool structs that are returned. (This will not release the instance of the vpgTool in the DB.)

Prototype Syntax:

```
vpgTool_t * vpg_tool_getByName(char *tName)
```

86. vpg_tool_getByKey().

This routine returns the tool whose key is "tKey". The key field should be unique, but that is not this routine's responsibility. If none is found with key "tKey" then NULL is returned.

Returns: a pointer to a vpgTool_t, or NULL on failure.

Note: it is the calling routine's responsibility to free the vpgTool struct that are returned. (This will not release the instance of the vpgTool in the DB.)

Prototype Syntax:

```
vpgTool_t * vpg_tool_getByKey(char *tKey)
```

87. vpg_tool_getName().

This routine returns the NAME" field from a vpgTool object.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getName(vpgTool_t *tool)
```

88. vpg_tool_getRunHost().

This routine returns the "Run Host" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getRunHost(vpgTool_t *tool)
```

89. vpg_tool_getExecPath().

This routine returns the "Executable Path" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getExecPath(vpgTool_t *tool)
```

90. vpg_tool_getExec().

This routine returns the "Executable" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getExec(vpgTool_t *tool)
```

91. vpg_tool_getArgs().

This routine returns the "Arguments" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getArgs(vpgTool_t *tool)
```

92. vpg_tool_getRunDir().

This routine returns the "Run Directory" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getRunDir(vpgTool_t *tool)
```

93. vpg_tool_getGroups().

This routine returns the "Groups" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getGroups(vpgTool_t *tool)
```

94. vpg_tool_getKey().

This routine returns the "key" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getKey(vpgTool_t *tool)
```

95. vpg_tool_getOutput_Input().

This routine returns the "OUTPUT_INPUT" associated with a tool.

Returns: a pointer to a dynamically allocated string on success, NULL otherwise. Note: It is the calling routine's responsibility to free the memory pointed to by the return value.

Prototype Syntax:

```
char * vpg_tool_getOutput_Input(vpgTool_t *tool)
```

96. _vpg_tool_getToolListObject().

This routine returns the tool_list object from the DB. It currently only expects to find one.

Returns a pointer to the DB_Object on success, otherwise it returns a a NULL pointer.

NOTE: it is the calling routines responsibility to free the DB_Object that is pointed to in the return. Free it using db_free_DB_Object_struct().

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static DB_Object *_vpg_tool_getToolListObject()
```

97. vpg_db_open().

This routine manages connections to the VPG DB. It keeps a reference count of the number of times a connection is established.

Prototype Syntax:

```
int vpg_db_open(char *host,  char *prog,  int num)
```

98. vpg_db_close().

This routine manages requests to close a connection to the VPG DB.

Prototype Syntax:

```
void vpg_db_close()
```

99. vpg_numObjects().

This routine returns the number of objects in the list "obj_list".

Returns: an integer 0 or higher.

Prototype Syntax:

```
int vpg_numObjects(DB_Object *obj_list)
```

100. vpg_fileTransport().

This function is used for transferring files from one machine to another. It is a part of the VPGcore services.

"filename"	- Name of file to transfer (including path).
"host"	- Name of host to transfer to.

Returns 1 on success, -1 otherwise.

Prototype Syntax:

```
int vpg_fileTransport(char *filename,  char *host)
```

101. vpg_launchTool().

This routine is for starting up standalone programs that run externally to VPG. If 'state' is SYNC (as opposed to ASYNC) vpg_launchTool waits for the prog being launched to terminate before returning.

Launching tools is really the job of VPG Daemons. This routine determines which daemon is responsible for launching the tool (prog->Run_Dir) and then it sends a "launch" message to that daemon with all of the necessary info.

SYNC and ASYNC are defined in CORE/H/vpg.h.

Returns 1 on success, -1 otherwise.

Prototype Syntax:

```
int vpg_launchTool(int state, Prog_Info *prog)
```

102. old_vpg_launchTool().

This routine is for starting up standalone programs that run externally to VPG. If 'state' is SYNC (as opposed to ASYNC) old_vpg_launchTool waits for the prog being launched to terminate before returning.

SYNC and ASYNC are defined in CORE/H/vpg.h.

Returns 1 on success, -1 otherwise.

Prototype Syntax:

```
int old_vpg_launchTool(int state, Prog_Info *prog)
```

103. vpg_free_vpgMDL_t().

Frees the vpgMDL_t struct pointed to by ptr.

Prototype Syntax:

```
void vpg_free_vpgMDL_t(vpgMDL_t *ptr)
```

104. vpg_free_vpgScenario_t().

Frees the vpgScenario_t struct pointed to by ptr.

Prototype Syntax:

```
void vpg_free_vpgScenario_t(vpgScenario_t *ptr)
```

105. vpg_free_vpgScenModel_t().

Frees the vpgScenModel_t struct pointed to by ptr.

Prototype Syntax:

```
void vpg_free_vpgScenModel_t(vpgScenModel_t *ptr)
```

106. vpg_free_vpgTool_t().

Frees the vpgTool_t struct pointed to by ptr.

Prototype Syntax:

```
void vpg_free_vpgTool_t(vpgTool_t *ptr)
```

107. _vpg_extractHostName().

This routine extracts the hostname from a "PROGRAM_PATH" value. A "PROGRAM_PATH" value looks like "hostname:path", where 'hostname' is optional. Therefore, 'hostname' is anything that precedes a colon. If no colon is found, then there is no hostname.

Returns a pointer to the hostname on success, or NULL if no hostname is found.

Prototype Syntax:

```
char *_vpg_extractHostName(char *in_path)
```

108. _vpg_extractExecPath().

This routine extracts the pathname from a "PROGRAM_PATH" value. A "PROGRAM_PATH" value looks like "hostname:path", where 'hostname' is optional. The colon is optional if there is no hostname.

Returns a pointer to the pathname on success, or NULL if no hostname is found.

Prototype Syntax:

```
char *_vpg_extractExecPath(char *in_path)
```

109. vpg_makeKey().

Return a uniq string that can be used as an indexing key in the vpg database.

NOTE: it is the calling routine's responsibility to free the key.

Prototype Syntax:

```
char * vpg_makeKey( void )
```

110. vpg_makeKeyShowFormat().

vpg_makeKeyShowFormat() prints (to stdout) a table showing the known format(s) for (the) vpgKey(s). See also: vpg_makeKey();

Prototype Syntax:

```
void vpg_makeKeyShowFormat(void)
```

111. _str_substring_remove()

Replace big_string with all of big_string minus the contents of sub_string.

RETURN 1 upon success.
otherwise if sub_string was not found, or could not be removed, then RETURN 0

Prototype Syntax:

```
int _str_substring_remove( char *big_string, char* sub_string )
```

112. str_tolower().

char * str_tolower(char *s);

return the string pointed to by s converting UPPERCASE to lowercase.

Prototype Syntax:

```
char * str_tolower( char *s)
```

113. vpg_util_createProg_Info().

Mallocs and initializes a Prog_Info structure.

Prototype Syntax:

```
Prog_Info        * vpg_util_createProg_Info()
```

114. vpg_util_freeProg_Info().

This routine frees memory previously allocated for a Prog_Info structure.

Prototype Syntax:

```
void vpg_util_freeProg_Info(Prog_Info *prog)
```

115. vpg_printf_control();.

int vpg_printf_control(int msg_type, int on_off, FILE* destination);

An interface to allow applications to control VPG LIBRARY msg level & destination.

msg_type is one of:
VPG_ERR
VPG_WARN
VPG_STAT
on_off is one of:
0 (turns off reporting all msgs of msg_type)

1 (turns on reporting message of msg_type)

destination if not NULL, will redirect and write all messages of type msg_type to the file pointed to by destination. The default destination for all messages channels is stderr.

Returns 1 on success 0 on failure.

Prototype Syntax:

```
int vpg_printf_control( int type, int on_off, FILE*dest )
```

116. _vpg_err_channel_file().

Return file stream associated with “int channel” or stderr if none.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static FILE * _vpg_err_channel_file( channel )
```

117. _vpg_err_channel_string().

Return string associated with vpg (error) message channel "channel"
or the locally defined constant VPG_LIB_STRING if none.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static char * _vpg_err_channel_string( channel )
```

118. vpg_printf_fflush().

int vpg_printf_fflush(msg_channel)
channel is one of
VPG_ERR
VPG_WARN
VPG_STAT

Flushes the buffers for the channel associated with msg_channel,

On successful completion these functions return a value of zero. Otherwise EOF is returned. For fflush(NULL), an error is returned if any files encounter an error.

Prototype Syntax:

```
int vpg_printf_fflush( int c);
int vpg_fflush( int channel );  /* vpg_fflush is a synonym. */
```

119. vpg_printf().

int vpg_printf(msg_channel, fmt, args....)

Prints a message on a VPG Error channel. Defined default channels ("msg_channel") are:

VPG_ERR
VPG_WARN
VPG_STAT

fmt is standard formatted print format (see printf())
args are optional variable arguments for formatted print.

Return value:
mimics vprintf (returns #of characters transmitted or -1 on err).

On successful completion these functions return a value of zero. Otherwise EOF is returned. For fflush(NULL), an error is returned if any files encounter an error.

See also:

vpg_fflush() (vpg_printf_fflush()), vpg_printf_control()

Prototype Syntax:

```
int vpg_printf( int channel, char *fmt, ...)
```

120. _vpg_error_integrity_check().

static int _vpg_error_integrity_check(error_no)

do sanity checks on the argument "error_no".

returns:
0 if error_no does not make sense, or other errors.
1 otherwise.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static int _vpg_error_integrity_check( error_no )
```

121. _vpg_error().

void _vpg_error(int vpg_msg_num, char *addedmsg)

report a known message number to vpg error system for processing.

vpg_msg_num is the numeric id of the error.

addedmsg is an additional string of text to be printed along with the system default message. The system default message, (and addedmsg), are only printed when vpg_perror() is called.

if addedmsg == NULL, then just the system default message is printed (only after vpg_perror() is called).

An internal VPG system error handling function is called for each known error (vpg_msg_num).

Prototype Syntax:

```
void _vpg_error( int vpg_msg_num, char *addedmsg )
```

122. _vpg_err_handle_error().

static void _vpg_err_handle_error(Vpg_Msg_Id vpg_msg_num, char *moreinfo)

Called after testing for integrity of vpg_msg_num. this function prints the system msg associated with vpg_msg_num (see vpg_perror()) and it executes

The only difference between vpg_perror() and if moreinfo != NULL, then it is assumed to be a string msg and is printed to stderr.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static void _vpg_err_handle_error( Vpg_Msg_Id vpg_msg_num, char
*moreinfo )
```

123. vpg_perror().

void vpg_perror(char *s)

Print to stderr the last reported error (which was generated by a _vpg_error() call.

Prototype Syntax:

```
void vpg_perror( char *moreinfo )
```

124. _vpg_perror().

static void _vpg_perror_(Vpg_Msg_Id vpg_msg_num, char *moreinfo) just print the system error message associated with "vpg_msg_num". If moreinfo != NULL, then it is assumed to be a string msg and is printed to stderr.

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static void _vpg_perror_( Vpg_Msg_Id vpg_msg_num, char *moreinfo )
```

125. vpg_error_getErrno().

int vpg_error_getErrno(void);

Returns the integer value of the last error generated.

The integer returned is not the same as the values defined in the unix intro(2). This value should never be compare to a number (e.g. "3") Rather, compare the value with the enumerations defined in vpg_err.h

See vpg_err.c for a list of errors numbers and messages.

e.g.:

```
if( vpg_error_getErrno() == V_EISCONN )
    printf("socket in use");
```

Prototype Syntax:

```
int vpg_error_getErrno(void)
```

126. vpg_model_getListByGroup().

Returns a list of models (DB_Objects) that belong to one of the specified groups of models.

Prototype Syntax:

```
DB_Object * vpg_model_getListByGroup(char *group_name)
```

127. vpg_SL_addItem().

The VPG Symbol List library (vpg_SL) can be used to maintain lists of objects.

Adds an item (a DB_Object) to the list. The current list is checked to see if the new object is in it. If so the object is not added (duplicate objects cannot exist). Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: -1 on failure, 1 on success, 2 if item was already in list.

Prototype Syntax:

```
int vpg_SL_addItem(DB_Object *new_obj)
```

128. vpg_SL_removeItem().

Removes an item (a DB_Object) from the list. If the item being removed is the currently selected item, then the current selection is set to NULL. Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: -1 on failure, 1 on success, 2 if item was not in list.

Prototype Syntax:

```
int vpg_SL_removeItem(DB_Object *remove_obj)
```

129. vpg_SL_numItems().

This routine returns the number of items in the selection list. Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: the number of items in the list or -1 upon failure.

Prototype Syntax:

```
int vpg_SL_numItems()
```

130. vpg_SL_getItems().

Returns the list of items in the selection list. Sets the arg 'num' to the number of items in the list, or -1 on error. Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: a pointer to the list (represented by a linked list of DB_List structures) on success, or NULL otherwise. Note: It is the calling routine's responsibility to free the return list. Use db_freeDB_List().

Prototype Syntax:

```
DB_List * vpg_SL_getItems( int *num )
```

131. vpg_SL_getCurrItem().

Returns the current item from the list of items in the selection list. The current item is the item most recently picked, selected, highlighted, or otherwise designated. Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: a pointer to a DB_Object upon success, otherwise NULL.

NOTE: it is the calling routine's responsibility to free the DB_Object pointed to by the return, if it is not NULL.

Prototype Syntax:

```
DB_Object * vpg_SL_getCurrItem()
```

132. vpg_SL_getNthItem().

Returns the n'th item from the list of items in the selection list. Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: a pointer to a DB_Object upon success, otherwise NULL.

NOTE: it is the calling routine's responsibility to free the DB_Object pointed to by the return, if it is not NULL.

Prototype Syntax:

```
DB_Object * vpg_SL_getNthItem( int n )
```

133. vpg_SL_setCurrItem().

Sets the 'cur_selected_item' in a selection list. The current item is the item most recently picked, selected, highlighted, or otherwise designated. If 'set_obj' is not in the selection list an error is returned. To unset the item, 'set_obj' should be a NULL DB_Object pointer ((DB_Object *)0). Currently, this routine will find the list. Future version may be passed the list to work with.

Returns: 1 on success, -1 on failure.

Prototype Syntax:

```
int vpg_SL_setCurrItem( DB_Object *set_obj )
```

134. vpg_mdl_duplicateMdl().

This routine creates a copy of an "mdl" and returns a reference to the copy. The new mdl is created in the DB and referenced via a DB_Object structure.

Note: it is the calling routine's responsibility to free the DB_Object struct that is pointed to by the return. (This will not release the instance of the DB_Object in the DB.)

Prototype Syntax:

```
DB_Object * vpg_mdl_duplicateMdl( DB_Object *obj_to_duplicate )
```

135. _vpg_SL_getObj().

This routine returns the selection_list object from the DB. It currently only expects to find one.

Returns a pointer to the DB_Object on success, otherwise it returns a NULL pointer.

NOTE: it is the calling routines responsibility to free the DB_Object that is pointed to in the return. Free it using db_free_DB_Object_struct().

Prototype Syntax:

NOTE: Not callable as an API - used internally.

```
static DB_Object *_vpg_SL_getObj()
```

INTENTIONALLY LEFT BLANK

APPENDIX C

“VPG_COMMAND” LANGUAGE SYNTAX

INTENTIONALLY LEFT BLANK

"VPG_COMMAND" LANGUAGE SYNTAX

The vpg_command command line interface by default receives its input from an HTTP (world wide web) internet server. This is because it is linked to client libraries of the National Computational Science Alliance (NCSA) HTTP daemon (HTTPd), which provide this capability. Optionally, vpg_command can receive its input from the keyboard (or a file). Before operating vpg_command, the VPG database must be launched and running. (This is because vpg_command acts as a client to the VPG database server.) By default, vpg_command expects the database to be running on the same host; however (and in the spirit of VPG architectural requirements for distributed computing), an optional host may be specified with the "-d" command line option (see section **3.1.2.2.2 HTTP server**). The VPG database uses unique keys to identify data records externally. These keys can be any text string but are similar to "*KEYverSep96_385_0x803f279a_511_842878187_20462_2*" and are usually proceeded by a "KEY =" identifier.

This appendix displays examples of syntactical construction for the implemented commands recognized by vpg_command during Project Focus. It is not meant to be a rigorous definition of the vpg_command language, but we think that the simple examples are straight-forward enough to be self explanatory.

1. Define (build) a Scenario Model:

```
VPG_COMMAND = SCENARIOMODEL_DEFINE
BEGIN = SCENARIOMODEL_DEFINE
 NAME = an instance of a scenario model.
    #  the key that follows must be a TOOL's key (an MDL's key).
 KEY  = key_for_the_mdl
END = SCENARIOMODEL_DEFINE
```

2. Define (build) a Scenario.

```
VPG_COMMAND = SCENARIO_DEFINE
NAME = my scenario name
BEGIN = ADD_MODEL
   # scenario Model keys follow. these are added to the
   # scenario
  KEY = key1
  KEY = key2
  KEY = key3
END = ADD_MODEL
```

3. Delete a Scenario.

```
 VPG_COMMAND = SCENARIO_DELETE
BEGIN = DELETE
    # keys for scenarios to be removed follow.
   KEY = KEYverSep96_385_0x803f279a_511_842878187_20462_2
   KEY = KEYverSep96_385_0x803f279a_511_842878150_20459_2
   KEY = KEYverSep96_385_0x803f279a_511_842878033_20423_2
END = DELETE
```

4. Delete a Scenario Model.

```
 VPG_COMMAND = SCENARIOMODEL_DELETE
BEGIN = DELETE
    # keys for scenarios Models to be removed follow:
   KEY = KEYverSep96_385_0x803f279a_511_842878187_20462_1
   KEY = KEYverSep96_385_0x803f279a_511_842878150_20459_1
   KEY = KEYverSep96_385_0x803f279a_511_842878033_20423_1
END = DELETE
```

5. Delete a Tool (MDL).

```
 VPG_COMMAND = TOOL_DELETE
BEGIN = DELETE
    # keys for Tools (MDLs) to be removed follow:
   KEY = KEYverSep96_385_0x803f279a_511_842878187_20462_0
   KEY = KEYverSep96_385_0x803f279a_511_842878150_20459_0
   KEY = KEYverSep96_385_0x803f279a_511_842878033_20423_0
END = DELETE
```

6. List Tools. Lists all VPG TOOLs (MDL's) and their keys.

```
 VPG_COMMAND = TOOL_LIST
```

7. List Scenario Models. Lists all Scenario Models and their keys.

```
 VPG_COMMAND = SCENARIOMODEL_LIST
```

8. List Scenarios. Lists all Scenarios and their keys.

```
 VPG_COMMAND = SCENARIO_LIST
```

9. Execute a Scenario. (Run a given scenario or scenarios).

```
 VPG_COMMAND = SCENARIO_EXECUTE
BEGIN = SCENARIO_EXECUTE
   # keys for scenarios to follow
   KEY = KEYverSep96_385_0x803f279a_511_842878187_20463_0
END = SCENARIO_EXECUTE
```

INTENTIONALLY LEFT BLANK

APPENDIX D

ORIGINAL ARCHITECTURAL REQUIREMENTS

INTENTIONALLY LEFT BLANK

ORIGINAL ARCHITECTURAL REQUIREMENTS

The intent of the original requirements was to fulfill the perceived needs of the tester in a virtual environment. The requirements were generated by experienced testers, simulation/model builders, and users thereof. In addition to attempting to repair or alleviate the drawbacks of modeling and simulation of the past, the requirements list was developed to

1. Provide the capability to validate weapon system simulations;
2. Allow interoperability, interchangeability, and reuse of VPG elements and model components; and
3. Comply with DoD architectural standards and guidance as they develop.

These requirements are a mixture of high level functional requirements, technical objectives, and specific solutions to foreseen needs. The requirements are displayed as shaded text. Each requirement is justified by a rationale. Many of the requirements include a further explanation to clarify the requirement's meaning. This stated, we now present the list of VPG requirements preceding Project Focus.

ATC Required Capabilities for the VPG Architecture

1. The architecture shall be distributed.

Explanation: By distributed, it is meant that components of the architecture do not necessarily have to be bound to one machine. Users shall be able to access (remotely or otherwise) architecture procedures and services from various platforms. Additionally, the architecture shall not be specific to one hardware system (machine). That is, the architecture should not be so tightly coupled to a particular vender's operating system and hardware that it cannot be ported to other operating systems of a similar nature. (For example, if the architecture operates in a "UNIX-like" operating system, it should reasonably portable to other "UNIX-like" operating systems.)

Rationale: The distributed nature of the architectures adds to the user's (tester's) flexibility and overall usefulness and attractiveness. Portability is important as it will allow the exploitation of computer hardware advances.

2. A) The architecture shall support distributed processing at the M&S level for those M&S applications that can benefit from distributed processing.

Explanation: Some M&Ss are a capable of subdividing their tasks and processing duties in a way that each portion can be processed by different (distributed) system resources. The architecture will be able to be used to start distributed portions of applications that are already distributed.

Another aspect is that the architecture shall be able to be used to start multiple simulations that are globally part of one exercise.

Rationale: Sometimes it is advantageous to use all the computing assets available (especially when real-time processing is necessary). If a simulation has distributed processing capabilities, the architecture should be able to take advantage of this capability.

See also: requirements 15, 17d, 18

2. B) The architecture shall support load balancing.

Explanation: Load balancing is when computer system resources (especially CPU usage) are balanced among processors in the distributed environment.

Rationale: Efficiency - especially to support real-time simulations.

Notes: Each simulation may or may not have internal load balancing. (For example, ModSAF will balance CPU load among platforms in an exercise with other platforms that are also running ModSAF.) In cases such as this, the architecture cannot do much to prevent (or support) load balancing. However, a global capability to control CPU resources of remote systems should be supported.

3. A) The architecture shall allow simultaneous multiple simulations.

Explanation: The architecture needs to be able maintain control of multiple simulations concurrently executing. Each of these simulations may consist of numerous individual models accessing a variety of data resources. The resources and models may be configured and maintained by various users of the VPG. It is important that individual user ownership of data, model, and other resources be maintained. Individual users must be able to set access permissions and restrictions for all owned resources. These restrictions must be strictly enforced.

Rationale: It is expected that multiple simulations will be run concurrently by multiple test directors (VPG users). Executing of simulations should not (except by design) intrude, interrupt, or otherwise interfere with the other executing simulations.

3. B) These simulations may be stand-alone or an integration of individual models running on different machines with independent or synchronized simulation clocks.

Explanation: "Stand-alone" means the simulation seeks and accesses its own input data, initializes and writes to its own output destinations, and otherwise communicates with no other "simulation."

Rationale: All simulations have a way of maintaining their internal sense of "time." Models interacting with each other absolutely must maintain some form of simulation clock synchronization. We think that this requirement implies that the architecture itself should have available some form of simulation synchronization timing service.

4. The architecture shall allow the inclusion or replacement of models and simulation components without having to rebuild the architecture core program.

Explanation: "Rebuilding" the architecture core implies major modifications of the fundamental source code that manages the architectural services, or even minor changes that require the architecture to be shut down, re-compiled, and replaced with the newly rebuilt version. Such modifications should only occur during distribution revision updates of the VPG architecture. If they are required whenever a model is incorporated into the VPG, then there is something fundamentally incorrect with the architecture's design. It is conceivable that occasionally minor modifications of an architectural sub-process might help integrate a simulation to make it compatible with the architecture or other simulations using the VPG, but this should be the exception rather than the rule.

Rationale: It is expected that multiple users will be accessing the architecture simultaneously. It will likely be an unbearable burden on some user's schedule if other users are shutting down the VPG, recompiling and re-installing it, in order to get their simulation "up and running." Furthermore, users should not have to be burdened with the detailed understanding of the architectural internal workings normally required to recompile and install the fundamental architecture. Furthermore, incorrectly installing or configuring a multi-user system (such as the VPG architecture) could lead to security holes, effectively corrupting or destroying the hard work of others. This type of operation should be a strictly privileged access procedure, not executable by the casual user.

5. A) The architecture shall provide a scripting interface to construct and execute simulations.

Explanation: "Scripting" is a means of keyboard input that includes typing according to a defined syntax language the desired commands and having those commands executed interactively or in "batch" mode. A "scripting interface" is a tool that will make scripting easier by doing providing services such as syntax checking, on-line help, etc. In addition to the graphic user interface detailed in Requirement 17, a scripting mechanism should be available which duplicates many (if not all) services available through the GUI.

Rationale: Once familiar with a syntax language for executing typically available procedures via architectural services, many users find it convenient to build scripts. Numerous GUI steps can be compiled into scripting files. These script files may facilitate executing large batch jobs during which, parametric variations are conducted on simulation initial conditions.

5. B) This interface shall facilitate automated simulation communication.

Explanation: Data elements, inputs, outputs, should flow freely form one simulation to another. The mechanism by which this occurs should be selectable by the user or left to the architecture to decide which is the best means to transport the data. For example, the user might specify that two simulations (which are capable of doing so) shall be communicating on a particular networking port via UDP[2] . On the other hand, the user might just specify that certain outputs from one model are to be used for certain inputs to another model and let the architecture decide how to conduct the data transport.

Rationale: This is a convenience to the implementer who is installing a virtual test scenario on the VPG. It adds flexibility by allowing a variety of communication means (those supported by the architecture and those not).

6. A) The architecture shall not prevent a simulation application from directly interfacing to other simulation applications or resources (as may be necessary for performance requirements).

Explanation: Communication means not supported by the architecture by means of a convenience service, but for which simulations are capable, shall not be purposely blocked by the architecture unless the communication interferes with the architecture's stability/integrity.

Rationale: This is a convenience to the implementer who is installing a simulation system on the VPG. It shall minimize the time required to install some simulations--at the possible cost of future simulation component re-use.

6. B) The architecture shall be capable of maintaining a description of the interface, even if the ability to monitor or control the interface cannot be provided.

Explanation: In a well-documented model that is highly integrated into the VPG, all its required interfaces will be documented (described) in some manner. The architecture should support the description of all INPUT/OUTPUT requirements--even those requirements that specify heretofore unknown data types or transport medium.

Rationale: A description of the model's interfaces is vital and necessary to assist in a model's re-use. Users not familiar with a particular model could be aided if the architecture can inform them about particular interface requirements of that model, even if the architecture cannot support those requirements. The ability to describe as yet unknown data objects and communication means is important for further explanation of VPG capabilities.

7. A) The architecture shall be capable of external communication by means of TCP/IP and DIS protocols.

[2] UDP - Universal Datagram Protocol.

Explanation: This means that convenient procedures shall be available, which allow simulations to use these protocols.

Rationale: These are popular simulation communication protocols.

7. B) Additionally, the architecture shall be capable of providing communication services through RS232/RS422 serial ports.

Explanation: Convenient procedures shall be available, which allow applications to use these input/output media.

Rationale: These are popular communication protocols especially for data collection, hardware in the loop (HIL), and instrumentation monitoring.

7. C) Both SLIP/PPP connections and simple modem protocols shall be supported.

Explanation: Convenient procedures shall be available, which allow applications to use these network extension tools.

Rationale: This will expand the availability of VPG services to computers and workstations not directly connected to the network on which the VPG is operating.

8. The architecture shall support H.320 video conferencing and T.120 multi-point data conferencing.

Explanation: These protocols are transparent to a TCP/IP network when bundled in internet protocol packets under such systems as the Virtual Internet Backbone for Multicast IP (MBONE). This does not imply that required hardware (cameras, microphones, video cards, etc.) are supplied any more than the CPUs, disk spaces, memory etc., required to run the base VPG architecture are supplied. It merely means that if the recommended and supported hardware and host environment are available, then they will be supported by VPG "tools" to access these items or capabilities (in this case, video conferencing).

Rationale: Building simulations (and even analyzing existing simulations) is often a collaborative effort. Video conferencing (VTC) is a tool that can be used to support these efforts. Furthermore, VTC is one of the more taxing activities on a network's bandwidth. If the network has the capacity to support VTC, then it should surely be able to support other (less taxing) activities, such as marker boards, audio conference sessions, etc.

9. The architecture shall provide a transparent and seamless I/O interface.

Explanation: Transparent and seamless refers to the quality that not much "human" intervention is required when the input and output interfaces for a simulation are sufficiently described for the architecture. For example, given that the architecture is able to execute and manipulate two models "**A**" & "**B**" selection of inputs and outputs. Suppose that model **B** requires terrain of a specified format as an input. Suppose that model **A** can provide three terrain databases, each of which satisfies model **B**'s requirements, then the architecture may require the user to select from one of these three databases. After that, the architecture "takes care of everything else." That is, because the rest of the things needed to be done in order to execute each model have been sufficiently described, the architecture is able to execute **B** transport its terrain output to (*where* and *when* it is needed by) **A**.

Rationale: Ease of use and configuration "sanity checking." The architecture could warn when unsupported input types or formats are about to be force fed to a simulation.

10. The architecture shall provide on-demand, transparent data import and export facilities for data, models, and terrain databases.

Explanation: This refers to specific tool sets or library procedures that are specific to the three mentioned areas: **data**, **models**, and **terrain databases**. **Data** can be thought of as primarily input providers (although the provider itself might be a model or simulation). **Models** primarily refer to simulations (which normally require some form of **data**). **Terrain databases** are chosen because they are often very central to ground vehicle testing (which is a primary focus of ATC).

Rationale: It is expected that these areas will require specialized support or will benefit from such support.

11. The architecture shall accommodate multiple levels of fidelity of models and terrain databases.

Explanation: **Models** will have an inherent fidelity level associated with them according to certain criteria. Thus, when assembling a test scenario, the tester will be able to choose from among higher and lower fidelity simulations, which all model the same phenomena. This will help the tester to conduct a reasonable balance of simulation strengths in order to apply the proper trade-offs to meet his or her needs.

Rationale: Simulations and data come with a broad variety fidelity trade-offs, to say the least. The architecture should not only adjust to this fact but take advantage of it.

12. The architecture shall support hardware in the loop (HIL) simulations concurrently with software simulations.

Explanation: Many of the synchronizing controls needed to allow concurrent communication between disparate simulations are the same (though their physical interfaces may vastly differ). Therefore, it should not matter to the architecture whether a model is a software simulation, HIL, or even a simple data file. Passing data from and to these models when and where the data are needed requires similar high level control procedures. The architecture shall supply these required procedures that will support HIL.

Rationale: Hardware systems are often used for testing purposes, both to provide stimulus for other devices and as the test items themselves. Interfacing a physical (hardware) device with software simulations is a common practice and can take advantage of the strengths of both types "models."

13. A) The architecture shall incorporate an MDL. *(Note. The MDL is intended to provide a description of a model of simulation usable by the VPG architecture and other simulation environments [including stand-alone applications] and to serve as a bridge between the architecture and these environments).*

Explanation: In order for the architecture to provide the types of services we have already mentioned, the architecture needs to know a lot about the objects (models or data) it intends to service. Some of things it needs to know are what the object can "do" (model output), what it "needs to do it" (model input), what is the object's intended purpose, and perhaps what are the strengths and weaknesses of the object. This knowledge and the ability to apply it to achieve the functionality which has been described (Requirements 4, 9, 10, 11, 12, 13, 14) is what is really required. The "MDL" is just a particular implementation of this concept (its mention does not really belong in a list of requirements). The other section of Requirement 13 (13B, 13C, 13D, 13E) affirms the type of qualitative information that should be retrievable using such a concept. Any other concept that can fulfill the vision of this concept is a likely candidate to satisfy this requirement.

Rationale: A means by which the architecture can achieve the stated objectives and services needs to be implemented.

13. B) The MDL shall also be capable of providing detailed hardware and resource requirements.

Explanation: See Requirements 15, 11.

Rationale: Intended to allow intelligent test scenario assembly.

13. C) The MDL shall also provide known out-of-range parameters for the output data.

Explanation: A model (or simulation) is just that. It is a (hopefully faithful) reproduction of the original. More precisely, it is an imperfect replication of *certain* aspects of the original. Normally, only the aspects of interest are modeled. It is usually impossible, and not necessary, to model the phenomenon's whole universe. Inseparable from the model are assumptions (implicit or explicit) made concerning known (and as yet unknown) factors that can impact the behavior or process being modeled. When internal or external parameters stray beyond the limits of these assumptions, the behavior of the model may become less faithful than otherwise. Indeed, the model may become unstable and fail catastrophically.

Ideally, all initial "safe operational" states should be known. If known, they should be described in some manner (MDLs or otherwise). However, this would be the exception; instead, it is more likely that only certain conditions are known which will cause the model to behave out of range of its intended purpose (and most faithful replication of the modeled phenomena. In either event, there should be a means by which safe (or unsafe) operating conditions can be documented and used in a way that facilitates re-use of the models and/or its components.

Rationale: Intended to facilitate error checking and boundary conditions. Designed to help during the process of interconnecting various models (building them into a test scenario).

13. D) The MDL shall be available for review and editing as human readable text.

Explanation: The intent here is that whatever the means used to describe models (their intended purpose, parameters, boundary conditions, fidelity limits, and the like), it shall be understandable by a person wishing edit or review it. It makes sense to use a "text"-based language syntax to model traits. This is because the text can then be edited by hand. As a future alternative, a GUI or other editor's assistant tool could be built on top of this base language.

Rationale: Intended to facilitate debugging data/model descriptions.

13. E) The MDL shall provide the necessary descriptions of models at varying levels of fidelity/level of detail to address problems relative to "uniqueness of models" and "too much detail."

Explanation: See Requirements 11 & 15. Uniqueness of models? What is that? "Too much detail" means too much fidelity is provided in areas where the user's test scenario does not require it.

Rationale: Intended to allow intelligent test scenario assembly.

14. A) The architecture, upon receipt of a "save simulation" command shall save the state of the simulation and issue a save simulation command to all external computing platforms that are part of the overall simulation.

Explanation: This is stated as a specification. However, the intent is to require the architecture to support a capability whereby one may maintain some form of control of remote components (presumably during a simulation exercise). At its most basic level, this reverts to the ability to start, stop, and save, load, and resume a loaded simulation. Note. The requirement is that the "command" is issuable. This means that the architecture sends the command only. The simulation must then correctly execute that command.

Rationale: The architecture should have a built-in support facility to coordinate, execute, save, and resume the distributed simulations of a test scenario. This is because it is often necessary to suspend or stop a test. This is particularly true when integrating virtual and "live" systems (or other HIL). Coordinating starts and stops in this case is often non-automated and it should not be overly burdensome to implement. Even when not integrating the live and virtual worlds, it is often necessary to save the state of a simulation. Of course, being saved is of little use if the saved state cannot be reloaded into the model and the simulation resumes at that point.

14. B) The architecture shall be able to load and restart a saved simulation to include issuing reload and restart commands to external computing platforms.

Explanation: See 14A.

Rationale: The architecture should have a built-in support facility to coordinate, execute, save, and resume the distributed simulations of a test scenario.

14. C) The restart of each simulation component shall be capable of being relegated to a different computing platform than was executed before the save simulation command.

Explanation: See 14A.

Rationale: The architecture should have a built-in support facility to coordinate, execute, save, and resume the distributed simulations of a test scenario.

15. The architecture shall support the capability of a dedicated platform (Resource Manage, RM) for the resource use optimization of connected resources. *(Notes: That is, requesting architectures could communicate with the RM if one existed rather than with individual external instantiations of the architecture. The RM and the services that it provides are not, however, a necessary component for the architecture to perform as required herein.)*

Explanation: This relates to the load balancing (Requirement 2). The RM is a means to perform efficient distribution of resources. If a test director is defining a test scenario "off line," he or she has no knowledge whether a certain resource will be available at the time the test is conducted. The RM will be able to arbitrate resource contention and the most optimal use of resources at the time the test scenario is conducted.

Rationale: Efficiency - especially to support real-time simulations.

16. The architecture shall support the connectivity and integration to documentation, analytical, and data visualization services for M&Ss.

Explanation: There are a large number of superior documentation tools (word processors, spread sheets, drawing programs) analytical tools (statistics packages, database management system [DBMS], engineering tools) and data visualization programs/libraries (2/3D plotting programs, GIS, polygonal and solid modeling environments). It would be wasteful to attempt to produce lesser quality duplicates of these available capabilities. The architecture should have "hooks" built in to attach and (at least partially) integrate such utilities.

Rationale: Tools such as these are likely to be used in any event. It makes sense to integrate them into the environment as much as is practical.

17. A) The architecture shall support a highly user-oriented interactive GUI to construct, execute, and monitor simulations.

Explanation: GUIs are nice for some people.

Rationale: Casual users will likely find the VPG more accessible if those procedures most often executed are presented in a GUI environment. Casual and new users will be able to immediately produce useful work without having to know every capability or memorize a particular syntax.

17. B) The interactive interface shall provide a display of the logical connectivity of the overall simulation under the control of the resident architecture.

Explanation: Most people gain a better concept of simulation connectivity by looking at a pictorial map of that connectivity rather than by lengthy listings or textual descriptions of the same. The architecture should show those unions (both before and during test scenario execution).

Rationale: One of the primary procedures foreseen for the VPG is the operation of piecing together simulation components or modeling them in a test scenario mosaic. Therefore, it is important to present the user with as clear a picture of the couplings as possible.

17. C) The interface shall be capable of monitoring and recording the real-time performance and execution of all model-level components of a simulation including resource usage (CPUs, processes, disks, instrumentation, etc.)

Explanation: See Load balancing #2.

Rationale: The VPG user should be able to decide (when appropriate) what systems will be used to run certain applications, especially in the event that some systems would otherwise be overburdened. These stated metrics (**CPUs, processes, disks, instrumentation)** and other system resource measurements are indicators of a system's overall capacity to accept more processing burden.

18. The architecture shall support resource contention arbitration (i.e., with respect to models, data, etc.) by means of user-selectable criteria to include first come, first served; round robin time sharing; or a user-designated priority system.

Explanation: The intent is for the architecture to support resource contention at some level by some method(s). By their nature, certain devices and resources can only process service requests serially.

Rationale: It is expected that multiple simulations will be run concurrently by multiple test directors (VPG users). The architecture must therefore support some form of handling this situation in a reasonable manner.

APPENDIX E

ACRONYMS AND TERMS

INTENTIONALLY LEFT BLANK

ACRONYMS AND TERMS

AML	ARC/INFO macro language
API	application programmer's interface
ARL	U.S. Army Research Laboratory
ASCII	American standard code for information exchange
ATC	U.S. Army Aberdeen Test Center
AUSA	Association of the United States Army
CGI	common gateway interface
CLI	command line interface
COTS	commercial off-the-shelf technologies
DB	database
DBMS	database management system
DFB	distributed fact base
DIS	distributed interactive simulation
DWB	designer's workbench
DXF	drawing exchange format
GIS	geographic information system
GNUplot	(not an acronym) a computer application for drawing 2D and 3D graphics
GUI	graphic user's interface
HEAT	high explosive antitank
HIL	hardware in the loop
HTML	hypertext markup language
HTTP	hypertext transport protocol
I/O	input and output
IP	internet protocol
KE	kinetic energy
M&S	models and simulations
MDL	model description language
ModSAF	modular semi-automated forces
PPP	point-to-point protocol
SLIP	serial line internet protocol
TCP/IP	transmission control protocol (over) internet protocol
TDB	terrain database
TDS	terrain database server
TECOM	U.S. Army Test and Evaluation Command
TIN	triangulated irregular network
TOPs	test operations procedures
TPA	technology program annex
UDP	user datagram protocol
UNIX	(not an acronym) an operating system
VPG	virtual proving ground
VTC	video teleconference
VV&A	verification, validation, and accreditation
WWW	world wide web

INTENTIONALLY LEFT BLANK

NO. OF COPIES ORGANIZATION

2 ADMINISTRATOR
DEFENSE TECHNICAL INFO CENTER
ATTN DTIC DDA
8725 JOHN J KINGMAN RD STE 0944
FT BELVOIR VA 22060-6218

1 DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL CS AL TA
RECORDS MANAGEMENT
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL CI LL
TECHNICAL LIBRARY
2800 POWDER MILL RD
ADELPHI MD 207830-1197

1 DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL CS AL TP
TECH PUBLISHING BRANCH
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 US ARMY TACOM
ATTN AMSTA TR D (JOHN WELLER)
BLDG 215 MAIL STOP 157
WARREN MI 48397-5000

1 FISCHER FRANCIS TREES & WATTS INC
ATTN WEIQUN ZHOU
200 PARK AVENUE
NEW YORK NY 10166

1 HQDA
DAMO FDQ
DENNIS SCHMIDT
400 ARMY PENTAGON
WASHINGTON DC 20310-0460

1 CECOM
SP & TRRSTRL COMMCTN DIV
AMSEL RD ST MC M H SOICHER
FT MONMOUTH NJ 07703-5203

1 PRIN DPTY FOR TCHNLGY HQ
US ARMY MATCOM
AMCDCGT M FISETTE
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

NO. OF COPIES ORGANIZATION

1 PRIN DPTY FOR ACQUSTN HQS
US ARMY MATCOM
AMCDGA D ADAMS
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 DPTY CG FOR RDE HQS
US ARMY MATCOM
AMCRD BG BEAUCHAMP
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 DPTY ASSIST SCY FOR R&T
SARD TT T KILLION
THE PENTAGON
WASHINGTON DC 20310-0103

1 OSD
OUSD(A&T)/ODDDR&E(r) J LUPO
THE PENTAGON
WASHINGTON DC 20301-7100

1 INST FOR ADVANCD TCHNLGY
THE UNIV OF TEXAS AT AUSTIN
PO BOX 202797
AUSTIN TX 78720-2797

1 DUSD SPACE
1E765 J G MCNEFF
3900 DEFENSE PENTAGON
WASHINGTON DC 20301-3900

1 USAASA
MOAS AI W PARRON
9325 GUNSTON RD STE N319
FT BELVOIR VA 22060-5582

1 CECOM
PM GPS COL S YOUNG
FT MONMOUTH NJ 07703

1 CPS JOINT PROG OPC DIR
COL J CLAY
2435 VELA WAY STE 1613
LOS ANGELES AFB CA 90245-5500

1 ELECTRONIC SYS DIV DIR
CECOM RDEC
J NIEMELA
FT MONMOUTH NJ 07703

NO. OF COPIES	ORGANIZATION
3	DARPA L STOTTS JPENNELLA B KASPAR 3701 N FAIRFAX DR ARLINGTON VA 22203-1714
1	SPCL ASST TO WING CMNDR 50SW/CCX CAPT P H BERNSTEIN 300 0'MALLEY AVE STE 20 FALCON AFB CO 80912-3020
1	USAF SMC/CED DMA/JPO M ISON 2435 VELA WAY STE 1613 LOS ANGELES AFB CA 90245-5500
1	US MILITARY ACADEMY MATH SCI CTR OF EXCELLENCE DEPT OF MATHEMATICAL SCI MDN A MAJ DON ENGEN THAYER HALL WEST POINT NY 10996-1786
1	APPLIED RESEARCH ASSOCIATES INC ATTN MR. ROBERT SHANKLE 219 W BEL AIR AVENUE SUITE 5 ABERDEEN MD 21001
1	SFA INC FREDERICK MFG DIVISION ATTN RICHARD MARVEL 20 S WISNER ST FREDERICK MD 21701
1	DIRECTOR US ARMY REDSTONE TECH TEST CENTER ATTN STERT TE C (CHARLES CROCKER) REDSTONE ARSENAL AL 35898-8052
1	COMMANDER US ARMY DUGWAY PROVING GROUND ATTN STEDP TD SP (DR LYMAN CONDIE) DUGWAY PROVING GROUND DUGWAY UT 84022
1	COMMANDER US ARMY ELECTRONIC PROVING GROUND ATTN STEWS EPG TT (DANIEL SEARLS) FORT HUACHUCA AZ 85613-7110
1	US ARMY AVIATION TECH TEST CENTER ATTN STEAT CO (LARRY EAGERTON) FORT RUCKER AL 36362-5276

NO. OF COPIES	ORGANIZATION
1	COMMANDER WHITE SANDS MISSILE RANGE ATTN STEWS IDD R (MARIO CORREA) WHITE SANDS NM 88002
1	PET COORDINATOR (VIRGINIA TO) PROGRAMMING ENV AND TRAINING 939 I BEARDS HILL ROAD SUITE 193 ABERDEEN MD 21001
1	NE PARALLEL ARCHITECTURES CENTER ATTN DR GEOFFREY FOX 111 COLLEGE PLACE SYRACUSE UNIVERSITY SYRACUSE NY 13244-4100
1	MS MARIE M G DERICO COMPUTER SPEC/RSCH ANALYST US MILITARY ACADEMY OFC OF ARTIFICIAL INTELLIGENCE ANALYSIS & EVALUATION WEST POINT NY 10996
1	TMAS BLDG 345 SOUTH (DAVID EDMISTON) SSAE ASM TMA PA PICATINNY ARSENAL NJ 07806-5000
1	US ARMY TACOM ATTN AMSTA TR S (TIM BAILEY) MAIL STOP 157 WARREN MI 48397-5000
	ABERDEEN PROVING GROUND
2	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LP (TECH LIB) BLDG 305 APG AA
1	COMMANDER HQ TECOM ATTN AMSTE TM T (DR DAVE BROWN) RYAN BLDG
1	COMMANDER HQ TECOM ATTN AMSTE CD (RAYMOND G POLLARD III) RYAN BLDG

NO. OF COPIES	ORGANIZATION
5	COMMANDER US ARMY ABERDEEN TEST CENTER ATTN STEAC CO (COL BAILER) STEAC TD (JAMES FASIG) STEAC TE (HARRY CUNNINGHAM) STEAC TE F (PAUL OXENBERG) STEAC RM C (ANDREW MOORE) BLDG 400
20	US ARMY RESEARCH LABORATORY ATTN AMSRL WM P MAJ M SMITH AMSRL WT WE J LACETERA AMSRL WT WE G SAUERBORN (15 CYS) AMSRL WT WE T R PERKINS AMSRL WT WE P CORCORAN AMSRL IS CS DR A MARK BLDG 120
8	US ARMY ABERDEEN TEST CENTER ATTN STEAC TE F A SCRAMLIN STEAC TE F R GAUSS STEAC TE I J SCHIMMINGER BLDG 400

INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1997	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Project Focus: A Study of Virtual Proving Ground Software Architecture Requirements	5. FUNDING NUMBERS PR: 1L1622618AH80
6. AUTHOR(S) Sauerborn, G.C. (ARL); Smith, K.G. (ARL); Scramlin, A.W., Shankle, R.R., Gauss, R.W. (ATC); Zhou, W. (CSC); Perkins, T.R., Corcoran, P.E. (ARL); Weller, J. (TACOM); Marvel, R.W. (SFA); Schimminger, J.P. (ATC)	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory Weapons & Materials Research Directorate Aberdeen Proving Ground, MD 21010-5066	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory Weapons & Materials Research Directorate Aberdeen Proving Ground, MD 21010-5066	10. SPONSORING/MONITORING AGENCY REPORT NUMBER ARL-TR-1429
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The virtual proving ground (VPG) is a concept being developed within the U.S. Army Test and Evaluation Command to harness the power of state-of-the-art sophisticated modeling and simulation technologies to augment and enhance test and evaluation in support of product acquisition. VPG is a cohesive and comprehensive capability for testing concepts, virtual prototypes, hardware prototypes, subsystems, and full systems. A broad, far-reaching, and diverse set of capabilities is envisioned within the VPG. Critical to the successful implementation of the VPG is an architecture able to support or enable those capabilities. A major function of the VPG architecture will be to integrate dissimilar heterogeneous engineering level models and simulations of prototype and production hardware and the synthetic environments in which they operate.

In 1996, the U.S. Army Aberdeen Test Center and the U.S. Army Research Laboratory jointly conducted "Project Focus" to help determine the architectural requirements that support the VPG concept. This report contains a description of Project Focus and the architectural requirements that resulted from it.

14. SUBJECT TERMS requirements, software reuse simulation, virtual proving ground software architechture			15. NUMBER OF PAGES 140 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102